

### 版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF





**broadview**  
www.broadview.com.cn

Android Internals: the Power User's View

# 最强Android书 架构大剖析



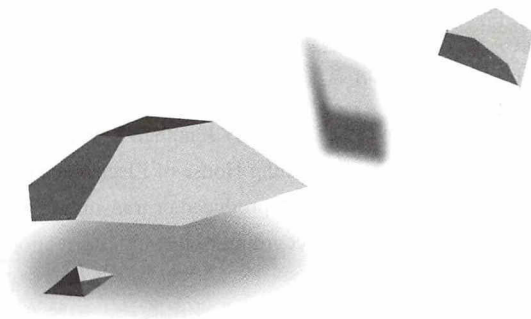
[美] Jonathan Levin 著  
崔孝晨 等译



中国工信出版集团



电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
http://www.phei.com.cn



Android Internals: the Power User's View

# 最强Android书 架构大剖析

[美] Jonathan Levin 著  
崔孝晨 等译

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING





## 内 容 简 介

本书通过实验而不是源码，将 Android 系统层层拆解，令读者深刻透彻地掌握 Android 系统的内部技术：以 init 进程为切入点详细阐述了 Android 的启动过程和关键服务；从 Android 作为资源协调者和服务提供者的角度，重点分析了 servicemanager 和 system\_server 这两个进程。同时，作者比较了 Linux 与 Android 系统的区别，并对 Android 系统的安全性做了深入的阐述。

本书采用了大量的图表示例和实验，表达新颖清晰，让读者能直观地掌握 Android 的技术精髓。本书适合广大移动开发者及对 Android 系统感兴趣的人员阅读。

Original English language edition copyright © 2015 by Jonathan Levin.

Chinese translation Copyright © 2018 by Publishing House of Electronics Industry.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission in writing from the Proprietor.

本书中文版专有出版权由 Jonathan Levin 授予电子工业出版社，未经许可，不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号 图字：01-2017-1884

## 图书在版编目（CIP）数据

最强 Android 书：架构大剖析 /（美）乔纳森·列维（Jonathan Levin）著；崔孝晨等译. —北京：电子工业出版社，2018.7

书名原文：Android Internals: the Power User's View

ISBN 978-7-121-31813-9

I. ①最… II. ①乔… ②崔… III. ①移动终端—应用程序—程序设计 IV. ①TN929.53

中国版本图书馆 CIP 数据核字(2017)第 130062 号

策划编辑：刘 皎

责任编辑：白 涛

印 刷：三河市双峰印刷装订有限公司

装 订：三河市双峰印刷装订有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787×980 1/16 印张：22.5 字数：468 千字

版 次：2018 年 7 月第 1 版

印 次：2018 年 7 月第 1 次印刷

定 价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888, 88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819, faq@phei.com.cn。



## 推荐语 |

这本书的确是目前一流的 Android 书。

——wushi (吴石), 腾讯科恩实验室负责人

一本对 Android 底层架构全面、深入剖析的书, 结合 Linux 有针对性地帮助读者从整体上把握 Android 架构的整体知识, 并对每个模块都做了十分详尽的解读, 帮助读者从细节上掌握每一个模块的要点。

——张鸿洋

也许你是刚入行的 Android 菜鸟, 也许你已经是有丰富经验的 Android 高工, 但是每个 Android 开发者都应该阅读一下这本书, 它会让你了解真正的 Android, 让你对 Android 底层系统有一个全新的认识。

——Stormzhang, 公众号: stormzhang

作者用“上帝”的视角, 向我们展示了一个 Android 系统的设计与架构, 庖丁解牛般地让读者无须接触大量源代码就能了解整个系统的实现思想, 而这是比源代码更加重要的东西。相信读者在这本书的指引下一定会对 Android 系统有更加深入的理解和认识。

——徐宜生, 《Android 群英传》作者





#### IV | 最强 Android 书：架构大剖析

这本书很适合用来学习、研究 Android 的系统架构。书中对比了 Android 与 Linux 系统，涵盖了文件系统，框架服务架构和安全等各个方面，为我们展现了具体且全面的 Android 系统的内部细节。此外，作者条理清晰，擅于将复杂的事情讲得简单透彻，显然造诣相当深厚。

本书可谓是了解 Android 系统内部技术的不二之选。

——段建华，技术小黑屋（[droidyue.com](http://droidyue.com)）博主，公众号：droidyue\_com



## 推荐序一 |

Android 是当今最主流的移动端操作系统,然而作为安全研究者要找到一本适合入门学习的书籍却并不容易。本人总结其原因有三:第一,Android 操作系统更新周期较短,特别是近两年 Android 自 4.4.X 更新至 7.1 版,它的系统安全特性已经发生了翻天覆地的变化,许多 Android 书籍自开始撰写到完工就需要几年时间,如果是英文书籍还涉及翻译的时间,通常读者拿到书的时候,内容已经比较过时了。第二,由于 Android 系统的复杂性,对作者的技术要求比较高。作者不仅要熟悉其原生(Native)层,对其 Java 层等组件也需要有所了解。市面上的很多 Android 书籍,很少能较好地覆盖每一个面向,或者是只有一个侧重点,这导致读者即便通读全书,也无法了解 Android 的全部。第三,很多书籍从 Android 源代码入手来讲解原理,虽有足够的深度,但略嫌乏味,会给读者一种纸上谈兵的感觉,给读者的阅读增添了困难,令他们很难全部读完并完全理解。

《最强 Android 书:架构大剖析》是我见过的 Android 书籍中,最适合安全研究人员阅读的一本。此书的作者 Johnathan Levin 和译者崔孝晨都是本人的朋友,相信与这两位打过交道的朋友都会发现,他们精力非常充沛,虽然已经从事安全研究十几年,但对新技术仍然充满了热情和好奇心。在面对面交流的时候,他们经常对着一个技术点侃侃而谈、乐此不疲。而作者 Johnathan Levin 更是在本书中引入“互联网思维”,为本书设立了网站 <http://newandroidbook.com>,并不定期撰写文章,听取读者的反馈和建议,把 Android 最新的、读者最想了解的特性分析更新到本书中。以上特性,保证了本书与那些“拿到就过时”的 Android 书籍相比,具有明显的优势。

一本好书,光与时俱进、有技术深度还远远不够,如何把一个复杂的操作系统的内部机制和原理,合理有序、循序渐进地传授给读者,也是一个需要推敲的问题,而这就不仅仅是要求作者技术功底深厚那么简单了。Johnathan Levin 多年从事技术培训工作,这些积累的培训经验确保了本书的易读性:细心的读者很快就能发现,本书的每一章节都相对独立,无论是顺序阅





读或者跳着看都没有太大的问题；书中大量使用图表、图片来叙述，让读者更直观地掌握各个知识点，并且通过实验的方式加深对各个知识点的印象，充分掌握一些比较重要的概念。而译者崔孝晨同志更是在确保把原书的含义完整无误地传授给读者的同时，加入了许多中国元素，在表达上更为生动形象——这样的合作，无疑是中国读者的一大福音。

Android 的安全防护机制是多维的，我的团队成员何淇丹、刘耕铭在 Mobile Pwn2Own 2016 中远程攻破搭载最新 Android 7.1 的 Nexus 6p 设备，从攻破所利用的漏洞来看，很明显，安全研究者需掌握 Android 浏览器、框架组件、内核等安全特性并找出 Java 层、Web 相关、原生层甚至内核层的漏洞，并串联在一起才能对 Android 进行有效的远程攻击、突破沙盒，最终实现提权。本书对 Android 安全特性的分析也是一大亮点，很好地覆盖了目前针对 Android 的攻击面。相信阅读本书一定会对您的工作有所帮助。

陈良 科恩实验室高级研究员

2018 年 6 月于上海



## 推荐序二 |

自 2008 年 Google 发布 Android 的第一版以来，时至今日，无论是在系统特性、用户规模还是生态规模上，它都取得了惊人的进展，获得了移动操作系统领域的绝对优势。Android 是开源的，这对于任何想要一探究竟的人都提供了非常大的便利，但同时由于 Android 系统本身日趋复杂，对大家也是个很大的挑战——一不小心就会陷进代码的汪洋大海之中。

Jonathan Levin 作为操作系统领域的专家，依赖自己深厚的技术功底和多年的研究，独辟蹊径地分别从高级用户和开发者的角度来探索 Android 系统。读者手上的这本书是从高级用户的角度开始 Android 的探索之旅的。这本书我首先接触的是英文版，现在非常高兴看到这本书的中文版面世，能让更多的读者受益。

本书特别适合高级用户（MIUI 称这部分用户为发烧友）学习使用。目前不少手机用户对各种硬件拆机评测很熟悉，这本书有如一个软件拆解，作者有如庖丁解牛一般，把运行在手机中的 Android 系统逐层拆解。在简要地介绍了 Android 的版本演化历史之后，作者先从分区和文件系统开始，详细介绍了各个分区的作用，各个分区上存储的内容和数据，还用实验详细演示了如何制作一个刷机包。在介绍了这些静态的软件组成之后，作者开始详细探索这些静态的内容是如何动态工作的。书中以关键的 init 进程作为切入点，详细阐述分析了 Android 的启动过程；接着分析了启动过程中的关键服务：原生服务和 Android 框架服务。操作系统有两个重要的角色：资源的协调者和服务的提供者。作者重点分析了 servicemanager 和 system\_server 这两个进程，它们构成了 Android 系统所扮演的两个角色的基石。

对 Android 系统有一点了解的读者可能知道，Android 是基于 Linux 内核的，那么 Android 和一个常用的 Linux 系统有何不同？作者接下来就从一个 Linux 用户的视角来观察和分析 Android 系统，剥去构筑在 Linux 内核之上的那层 Android 外衣，让一个熟悉 Linux 系统的人跃跃欲试：“我也能构建一个 Android 系统”。本书最后概要性地讲述了一些 Android 的安全机制，





虽然只有短短的一章，但是非常清晰，尤其是对 selinux 的描述。从上述的脉络可以看出，作者动静结合，抽丝剥茧一般把运行在手机里的 Android 系统清晰地展示在大家眼前。

本书虽然是从高级用户的角度来探索 Android 系统的，但也很适合 Android 开发者，尤其是 Android 系统工程师学习。要想剖析一个系统，得先了解使用它。这本书有如一盏指路明灯，让我们在 Android 代码的汪洋大海之中始终明确前进的方向。略有遗憾的是这本书来得有点晚，使我们在学习 Android 系统的过程中走过一些弯路，不过今天的读者可以幸运地站在大师的肩膀上了！在小米 MIUI，我们也打算使用其中的部分内容作为内部培训材料。如果您正好打开本书看到了这篇序，诚邀您一起开始我们的 Android 系统探索之旅，这将是一个妙趣横生的旅程。谢谢！

汪文俊 MIUI 系统平台部总经理

2018 年 6 月



## 译者序 |

市面上关于 Android 的书籍可谓汗牛充栋，我甚至都不敢把书名 *Android Internals* 按照惯例译为《深入理解 Android 系统》——重名的书太多了。那么为什么还要把这本书介绍给国内的读者呢？因为市面上绝大多数的 Android 书籍都是从程序员的视角展开的，入门的门槛相对较高。尽管开发 Android App 的程序员们自然应该对 Android 系统有一个深入的理解，但这并不意味着其他人并不需要理解 Android 系统。比如，电子取证人员，他们需要对 Android 中的文件系统及数据存放位置有一个清晰的认识，以便从中提取相关数据；喜欢折腾的技术发烧友，root 掉系统之后一般都喜欢自己修改一下系统，比如禁用一些开机启动项之类的。如果无须依赖额外的 App，只需一个文本编辑器就能完成相关修改，甚至给系统换上自己的开机动画岂不是很酷……诸如此类。但这些人中只有很少的一部分接受过正规的编程训练，因此市面上大部分的书籍对他们来说难度就太大了。

本书的作者 Jonathan Levin，也是畅销书 *Mac OS X and iOS Internals: To the Apple's Core*（中文版为《深入解析 Mac OS X & iOS 操作系统》）一书的作者。按 Jonathan 自己的说法，*Mac OS X and iOS Internals: To the Apple's Core* 一书的读者反馈中，反映最激烈的问题是：太技术化了！许多读者读起来感到头大！所以在这本后继的 *Android Internals* 中，他把不需要代码就能表达清晰以及与开发人员关系不太紧密的部分放在这一本书中，而把剩下的、与开发紧密相关的部分放在了另一本书中。这一点从本书英文版的副标题“*Power User*”就可以看出来。那么什么是“Power User”呢？如果一定要和传统的桌面系统的用户相对应，这个“Power User”就相当于系统管理员（administrator）的角色。相对于普通用户，他需要对系统有更加深入的理解，能对系统进行更加详细的配置，因而也被认为可以拥有较高的权限（比如 root）——本书的部分实验确实需要拥有 root 权限，且第 8 章中也专用了一个小节讨论 root 这一主题。

有人问，既然是讲系统内部实现，不讲编程又是如何把它讲清楚的呢？答案是使用实验。



本书的内容是根据作者多年讲课的讲义，整理、精选<sup>1</sup>而来，通过在 ADB（Android 调试桥）中执行各种命令的方式（相对于阅读代码），比较直观地向读者揭示 Android 内部的工作原理。效果如何呢？别看广告，看疗效。上次曝出的 CIA Value7 的相关内容显示，这本书已经被 CIA 私下盗版，用于 CIA 特工的内部培训了。而可怜的 Jonathan Levin 既不敢告 CIA 侵权，又不能告 WikiLeaks……，只好在本书官网上提供了已经被泄密的 2015 年 6 月版的英文版的免费下载链接——与其去 WikiLeaks 下载，不如上官网下载。不过读者也不必沮丧，自我开始本书的翻译以来，几乎每个季度都会收到 Jonathan Levin 发来的大量更新——其中包括历次 Android 系统更新的新内容，以及书中已经发现的一部分错误的更新（包括一些我发现的错误:))，使得我也不得不多次将译稿做一些必要的返工，目前出版的中文译本是以 2016 年 11 月底的最新版本为准（更新至 Android Marshmallow PR1 版）的，您大可不必担心白花银子。

在本书的翻译过程中，我们力求将原文准确、清晰地翻译成中文。有模糊不清之处，我们尽量通过与作者沟通、阅读源码和实验的方式搞清楚。但各类缩写还是本着忠实原文的原则，沿用原文的写法。如在本书中，Android JellyBean 版会被缩写成 J 版或 JB 版，Android Lollipop 版会被缩写成 L 版，Android Marshmallow 版会被缩写成 M 版等。

本书由上海公安学院的教师教官完成翻译，第 1 章由殷方老师翻译，第 2 章由王宏老师翻译，其余章节由我翻译，全书由我统一校对，并经本书作者 Jonathan Levin 及其国内合作培训公司的同志审校。

最后感谢电子工业出版社刘皎老师在本书翻译过程中给予我们的有力帮助，感谢腾讯公司科恩实验室吴石、陈良、赵泽光等老师给本书初稿提出的宝贵意见。

囿于译者水平有限，书中必然存在疏漏之处，敬请读者不吝指正。

崔孝晨

2018 年 6 月

---

1 确实是精选而来的，书中第 7 章的“wchan 和 syscall 文件”一节中有“你也可以使用上个实验中给出的方法，解析程序计数器（program\_counter）的值”字样，但之前的那个实验与这一节毫无关系。后经与作者确认，这里原本另有一个实验，在正式出版时删除，但是忘记相应地修改这里的语句，而遗留下了这个问题。

# 致中国读者 |

此刻没有什么事情比本书中文版的出版（经过几年的努力）更令我开心的了！在过去的几年中，上海已经成了我的第二故乡——我经常过来给开发者们培训 Linux、Android 和 MacOS 相关的内容。

本书的翻译历时较长（尽管还没有长到和本系列第二本书的写作一样）。在此过程中，Android 在全世界尤其是在中国的受欢迎程度与日俱增——华为、小米和其他手机厂商已经渐渐成为一流的智能手机生产商——他们打造了更好的设备并将这种能力扩展成一个丰富的生态体系。

我希望本书能帮助更多的人加深对 Android 系统的理解；我希望它能启发与操作系统交互时的创新的、激动人心的思考——令 Android 系统更优秀、更强大而且更有效率。

我本人非常乐意接受各种改进，所以请记住访问 <http://NewAndroidBook.com/>，让我知道你的所思所想！



# 目 录 |

关于本书 .....	XVIII
第 1 章 Android 体系结构的变革之路 .....	1
1.1 Android 系统版本的历史变迁 .....	2
Froyo (冻酸奶) .....	3
Gingerbread (姜饼人) .....	3
Honeycomb (蜂巢) .....	4
Ice Cream Sandwich (冰激凌三明治) .....	5
JellyBean (果冻豆) .....	5
KitKat (奇巧) .....	6
Lollipop (棒棒糖) .....	7
Marshmallow (棉花糖) .....	8
Nougat (牛轧糖) .....	9
1.2 Android 与 Linux.....	11
并非另一个 Linux 发布版本.....	11
然后 Android 就登场了.....	12
与 Linux 的异同 .....	13
Android 的框架.....	15
Dalvik 虚拟机 .....	18
JNI.....	19
原生二进制可执行文件 .....	20
Bionic.....	22
Android 的原生库.....	25
源自其他项目的原生库 .....	27
硬件抽象层.....	28

Linux 内核 .....	29
1.3 Android 的衍生产品 .....	30
谷歌官方的衍生产品 .....	30
非谷歌官方的衍生产品 .....	33
1.4 对前方道路的思考 .....	36
兼容 64 位 .....	36
ART (Android 运行时) .....	37
多画面 .....	38
把 Android 用作台式机操作系统 .....	38
Android 和 ARA 项目 .....	39
Brillo .....	40
本章小结 .....	40
参考文献 .....	41
第 2 章 Android 的分区和文件系统 .....	43
2.1 分区架构 .....	43
需要许多单独分区的原因 .....	44
GUID 分区表 .....	45
闪存 (Flash Storage) 系统 .....	46
文件系统 .....	46
Android 设备中的分区 .....	49
2.2 Android 文件系统中存储的内容 .....	53
root 文件系统 .....	53
/system 分区 .....	54
/data 分区 .....	65
/cache 分区 .....	71
/vendor 目录 .....	72
SD 卡 .....	73
2.3 受保护的文件系统 .....	74
OBB: Opaque Binary Blobs .....	74
ASec: Android 安全存储 (Android Secure Storage) .....	76
2.4 Linux 伪文件系统 .....	78
cgroupfs .....	78
debugfs .....	79
functionfs(/dev/usb-lfs/adb) .....	80
procfs(/proc) .....	81
pstore(/sys/fs/pstore) .....	81



selinuxfs(/sys/fs/selinux) .....	82
sysfs(/sys) .....	83
本章小结 .....	84
参考文献 .....	84
第 3 章 Android 的启动、备份和重置 .....	86
3.1 Android 系统镜像 .....	87
Boot Loader .....	89
Boot 镜像 .....	93
内核 .....	95
RAM disk .....	97
/System 和/Data 分区镜像 .....	99
3.2 启动过程 .....	101
固件启动过程 .....	101
内核启动过程 .....	105
3.3 关机和重启 .....	109
3.4 应用的备份和恢复 .....	112
命令行工具 .....	113
本地备份 .....	114
监视备份操作 .....	117
3.5 系统重置 (recovery) 和升级 .....	119
OTA (Over-The-Air) 升级包 .....	121
制作你自己的 ROM .....	124
制作 ROM 时可用的网上资源 .....	128
本章小结 .....	130
参考文献 .....	130
第 4 章 init .....	132
4.1 init 的角色和任务 .....	132
系统属性 .....	134
.rc 文件 .....	140
总结: init 的执行流程 .....	146
4.2 init 和 USB .....	150
4.3 init 的其他角色 .....	152
ueventd .....	153
watchdogd .....	154
本章小结 .....	154

本章讨论所涉及的文件 .....	155
<b>第 5 章 Android 的守护进程 .....</b>	<b>156</b>
5.1 core 类中的服务 .....	156
adbd .....	156
servicemanager .....	160
healthd .....	161
lmkd (Android L) .....	165
logd (Android L) .....	168
vold .....	173
5.2 网络相关服务 .....	182
netd .....	182
mdnsd .....	187
mtpd .....	187
racoond .....	188
rild .....	189
5.3 图形及多媒体服务 .....	190
surfaceflinger .....	190
bootanimation .....	192
mediaserver .....	194
drmserver .....	196
5.4 其他服务 .....	197
installd .....	197
keystore .....	200
debuggerd .....	204
gatekeeper (Android M) .....	207
sdcard .....	208
Zygote .....	211
本章小结 .....	214
本章讨论所涉及的文件 .....	214
参考文献 .....	215
<b>第 6 章 框架服务的架构 .....</b>	<b>216</b>
6.1 再探 servicemanager .....	217
6.2 服务调用的模式 .....	222
优点和缺点 .....	224
序列化和 Android 接口定义语言 (AIDL) .....	225

6.3	Binder .....	228
	简明历史 .....	228
	那么, Binder 究竟是什么 .....	229
	使用 Binder .....	230
	分析 Binder 的当前使用情况 .....	231
6.4	system_server .....	232
	启动及执行流程 .....	232
	修改启动时的行为 .....	234
	本章小结 .....	237
	本章讨论涉及的文件 .....	237
	参考文献 .....	237
第 7 章	从 Linux 角度看 Android .....	238
7.1	重温/proc .....	239
	符号链接: cwd、exe 和 root .....	240
	fd .....	243
	fdinfo .....	245
	status .....	247
7.2	用户模式内存管理 .....	254
	虚拟内存的分类和生命周期 .....	254
	内存的相关术语 .....	258
	内存不足时的应对方案 .....	266
7.3	跟踪系统调用 .....	269
	toolbox ps 工具 .....	269
	wchan 和 syscall 文件 .....	270
	strace 工具 .....	271
	本章小结 .....	272
	参考文献 .....	272
第 8 章	Android 安全性 .....	274
8.1	移动安全威胁建模 .....	275
	攻击向量 .....	275
	攻击之道 .....	278
8.2	Linux 层上的安全措施 .....	281
	Android 使用 Linux 权限的方式 .....	281
	Linux 权能 .....	289
	SELinux .....	294

其他值得注意的特性 .....	301
8.3 Dalvik 层上的安全措施 .....	305
Dalvik 层上的权限 .....	305
Dalvik 代码签名 .....	310
8.4 用户层上的安全措施 .....	312
锁屏机制 .....	312
支持多用户 .....	316
密钥管理 .....	318
证书管理 .....	318
密钥和私钥管理 .....	322
8.5 存储安全 .....	323
加密/data 分区 .....	323
基于文件的加密 (Nougat 7.1) .....	326
Direct Boot (Nougat 的新特性) .....	326
启动过程中加强验证 .....	327
8.6 Root Android 设备 .....	328
在设备启动环节中 root .....	329
利用安全漏洞 root .....	331
Root 对安全的影响 .....	332
本章小结 .....	334
参考文献 .....	334



# 关于本书 |

## 概览

购买了本书的朋友，毫无疑问你已经意识到了 Android 的重要性。这个启动于 2003 年的操作系统，在被谷歌收购之后，现在已经成为谷歌最得意的产品。它迎头赶上了苹果公司的 iOS 操作系统（也有人说是很接近了），不仅取得了移动操作系统领域内的绝对优势（截至本书付印时，它的市场占有率已经达到了令人惊异的 82% 了），而且还渗透到了其他平台上，成为可穿戴设备、TV 和嵌入式设备上的操作系统。

Android 是开源的，而且是可以免费获得的，这也就意味着任何人都能获得它，并对它进行修改，使之能够运行在任何一种平台上——事实上，这也是它能够力压群雄，占据市场主流的原因。不过，令人吃惊的是，尽管已被广为接受，但是至今仍然没有一本书来完成探究其内部工作原理并将其文档化的任务。前几年有一本名叫《构建嵌入式 Android 系统：移植、扩展和定制》<sup>1</sup>的书，作者是 Karim Yaghmour——这本书给出了大量关于该操作系统通用结构的细节信息，但是其着眼点在于如何创建和修改源码，使之能运行在各种新的平台上，而没能给出操作系统本身的结构。事实上，在此书“内部结构入门”一节中，Yaghmour 声称“要想完全理解 Android 系统服务的内部结构，无异于蛇吞象”。

我认为这还算是一种保守的说法，这也就是为什么本内容需要由好几本书组成而不是只有

---

<sup>1</sup> 该书的英文版书名是 *Embedded Android: Porting, Extending, and Customizing*，中文版书名为《构建嵌入式 Android 系统》（秦云川、肖淇译，中国电力出版社，2015 年 8 月出版），英文版书名中的“*Porting, Extending, and Customizing*”在中文版书名中没有译出，但为了方便读者理解下文，这里将其译出。——译者注

一本的原因。第 1 本（也就是你现在正在读的这本书）主要是从高级用户或者管理员的角度讨论 Android。在这一本中，我试图从各种不同的角度，如 Android 的设计、文件系统结构、启动顺序、原生服务再加上 Linux 基底以及 Linux 基底对操作的影响来讨论这一操作系统。这一切都不涉及代码，只是试图尽可能地给你一个大致概念和鸟瞰图。从某种程度上说，本书可以算作 Yaghmour 那本著作的后续之作，Yaghmour 的那本书本身也是极好的资料来源，我强烈建议你找一本来读。

本系列的第 2 本（将于不久后出版）将会对 Android 讨论得更深，而且会把视线转向 Android 框架服务（framework）的结构——这显然对开发者更有吸引力：通过使用 Java 层上各种丰富的框架，开发者可以拥有把输入设备、传感器、图形图像之类的东西抽象化的强大能力。当然这一抽象化的能力也并非是没有代价的——复杂性隐藏在“水面之下”，只是大多数开发者对此安之若素（更有甚者还满足于这一状态）罢了。不过知识是力量的源泉，深入熟悉各类框架（及其底层实现机制）对于任何想要进行底层开发或者在性能调优、支持更多的硬件、安全研究等方面有所建树的人士都是至关重要的。

Android 是一个不断飞速更新的系统。在本书开始编写时，最新版的 Android 系统还是 KitKat，然后（尽管中间有过几次跳票）最新版就变成 Lollipop 了。而且这一趋势还在不断加快——由于 Lollipop 版被发现有不少 Bug，谷歌又宣布将要推出 Android Marshmallow 版。不过，截至本书付梓时，Lollipop 版显示出了它稳定的一面——所以我也可以骄傲地说，这本书已经反映了最新版……好吧，是截至出版之日。幸运的是，借本书自媒体出版的东风，我可以不断地紧跟 Android 系统的更新而修订本书的内容，读者现在看到的这一版已经更新到 Marshmallow Preview Release 1 版（2015 年 6 月）了<sup>1</sup>。

我还试图从我的上一本书 *Mac OS X and iOS Internals* 中吸取一点“经验教训”。我收到的读者对那本书的主要批评之一是：那本书太技术化了，充斥着大量源码，非开发人员身份的读者读起来实在是太累了。我个人的信条是“读一下源码吧，淡定些！”——因为源码不像自然语言，（几乎）是不会有歧义的，因此也是用来描述系统的正确方式。虽然我还是坚持我的想法不动摇，但在这本书中，我还是在不牺牲细节信息的前提下，尽可能多地改用图表的形式来表达意思。[我把这一做法也用在了 *Mac OS X and iOS Internals* 一书的第 2 版上（这一版将于 2016 年下半年出版）——这倒也不完全是因为我想通了，心甘情愿地这么做，还有一个重要因素也在促使我这么做，那就是在那本书里更加深入地探讨了 Mac OS X 和 iOS 系统更底层、更隐秘的部

---

1 原文如此，就是苦了我这个译者，翻译的过程中不断地修订，原文都在不断地改，更何况译文……，简直是一个不断返工的死循环“哭”。截至本书中文版出版时，最新版为 2016 年 11 月底更新的版本，最新的内容更新至 Android Nougat 版。——译者注

分——这些东西可是没有源码的……]。

本书也十分强调动手实践，我从我们的 Android 培训课程里抽取了一些动手练习，并把它们改编成了书中的实验。如果你想要对相关章节讨论的主题有深入了解的话，这些实验对你来说无疑是极为重要的。Android 是 UNIX（实际上是 Linux）的一种衍生品，而学习 UNIX 的唯一正确方式是用我们的手指，而不是用我们的眼睛或耳朵。在这些实验里演示了 Android 命令行接口（CLI，command-line-interface）中的一些非常有用的命令，以及深入了解操作系统内部结构的技术。更进一步说，有些实验在不同的 Android 环境下会产生不同的输出结果——这也使它们非常值得你在自己的手机/平板电脑上亲手做一遍，以体验不同厂商或操作系统版本在架构和实现上的不同之处。

## 全书内容鸟瞰

本书的内容编排，既可以让你能按部就班地逐页阅读，也能让你随便翻开一页就能读下去。书中的每一章都是独立成章的，在你阅读本书的电子版时，文中所提及的相关主题都是以超链接的形式给出的——直接点击它，你就可以跳转到相关章节进一步阅读，但是对于纸质版的读者，我就只能给出相应的章节编号（引用书中的内容时）或者 URL（引用其他资料时）了。我也会在相应的地方附上所引用的 AOSP 文件的路径（尽管为了节省空间，是以缩略的形式给出的），要不然本书的主要用途就变成“防身”了……

第 1 章介绍了 Android 操作系统：介绍了它各个不同版本的演化史（从 Froyo 版开始，这是你目前在市场上能找到的最老版本的 Android 系统了，一直到 Lollipop 版<sup>1</sup>）。同时，在这一章里（从较高的视角）也通过逐一比较 Android 软件栈中的各个层（layout），阐述了 Android 的体系架构以及它的 Linux 基础。紧接其后，这一章还介绍了 Android 的衍生产品——既包括谷歌自己的，也包括其他厂商的（比如亚马逊的 FireOS）。最后，这一章将对 Android 未来发展方向的设想和思考作为整章的小结。

从第 2 章开始，我们开始深入探索各种技术细节——第 2 章的主题是 Android 的分区和文件系统。我们先讨论 Android 使用的分区架构（不幸的是，各家厂商远远没有对此达成一致），以及文件系统——EXT4 和 F2FS。然后，我们将探究文件系统中存放的内容——如果你要知道某个特定的系统目录或文件中存放的是什么数据，这将是非常有用的。此外，本章还会涉及一些内置应用的数据存放目录，如果你对电子取证感兴趣，这些内容无疑也是非常重要的。在这一章中还会讨论 Android 受保护的文件系统（OBB 和 ASEC）——尽管在系统被 root 之后，

---

1 事实上 2016 年 12 月的更新版中已经介绍到 Nuogat 版了。——译者注



这些保护措施就会失效。最后，我们还会阐述 Linux 伪文件系统（cgroupfs, debugfs, procfs, sysfs 等）在系统中扮演的角色。

第 3 章是在前一章的基础上展开讨论的——因为涉及分区。它解释了在 Android 系统启动过程中，各个分区所起的作用。我们先会讨论 Android 的启动镜像（尽管有时会有些不正确地把它称为 ROM），以及怎样把它刷到设备的各个启动分区里去。Android 默认使用的 Boot Loader 也会予以阐述（本书官网上还有一篇从更加技术的角度开展讨论的进阶阅读文章），以及启动镜像的其他一些组件〔内核（kernel）、设备树（device tree）和 initramfs〕也会被详细讨论。本章中的相关实验还演示了如何把这些组件从启动镜像中解出来，修改其中的内容，然后再把它们重新打包回去——制作一个你自己的刷机包（当然安装这种刷机包的前提是：在你的移动设备上 Boot Loader 已经解锁了）。此外，在这一章中还讨论了通过无线网络发送更新镜像进行（OTA）升级，以及设备备份、重置和关机的操作过程。

第 4 章专门讨论一个进程/init。这个进程和它在 UNIX 系统中的同名进程一样，是负责在用户态中启动系统的。我们会详细解释启动的过程，并解释/init.rc 文件中使用的语法。/init 的其他一些作用，比如维护系统属性和监视硬件改变（以 ueventd 进程的身份），也会详细地加以讨论。

在第 5 章中讨论的是原生服务，也就是列在/init.rc 文件中的，由/init 进程启动的 Linux 二进制可执行文件（与之相对应的是 Dalvik 级的框架服务，这些服务是以 system\_server 进程中的线程的形式被加载起来的，我们会在第 2 本中讨论这些框架服务）。在这一章中逐个详细介绍了你可以在自己的移动设备上找到的每一个守护进程——说实话，还真不少。

在第 6 章中简略介绍了 Android 框架服务的大致架构，解释了 servicemanager 和 system\_server 进程在其中所扮演的角色——这两个进程共同构成了其余所有构建在其上的 Android 框架服务的基石。Binder 也是这一章里的重头戏，我们会简略地对它进行一番描述，但是大部分细节信息还是要留待第 2 本讨论补充。我希望这一解释足以让你能更深一步地理解 Android 进程间通信和远程过程调用的内部工作机制。

第 7 章以 Linux 的视角来看待 Android，也就是通过/proc 伪文件系统以及使用 Linux 系统中的工具，观察 Android 系统中的进程及应用。这一章还有一个“一箭双雕”的作用——你可以把这一章讨论的绝大多数工具，用在你自己的 Linux 系统原生代码的调试工作中。

作为本书的最后一章，第 8 章是专门用来讨论安全的。这一章在本书的官网上有预览版（只不过在预览版中的编号中，它是第 21 章——当时我曾经天真地认为可以在一本书中把所有的问题都讲清楚），在这一章里将逐一详细讨论 Android 的所有安全特性——既包括 Linux 层上的，也包括 Dalvik 虚拟机层上的。同时，在这一章中还有一个小节专门来讲述 Android 设备的 root



问题——既讨论了“被厂商认可的”在设备启动时 root 的方式，也讨论了那些通过安全漏洞 root 设备的方法。

## 本书使用的排版约定

本书采用如下排版约定：

- 以 filename 这种格式表示文件名。
- 命令、系统调用名称以及框架类名都是以 command(1)、systemCall(2) 以及 classes 这种格式表示的。命令和系统调用名称后面跟的数字是指：在使用 Linux 的 man 命令打开的手册中，该命令或系统调用所在的章节编号。

此外，本书还有许多插图、代码清单和输出结果。插图是由系统组件或消息传递流程组成的图片，相对于输出结果，代码清单中给出的一般是内容固定的文件中的内容，而输出结果中给出的则是一连串命令的执行结果——它通常是某个实验的一部分。我制作输出结果的目的是：显示各条命令的执行顺序及其用法，所以输出结果中一般都是带注释的（如输出结果 0-1 所示）。

```
# Comment, explaining what's being done
user@hostname (directory) User input
Output...
Output.. # Annotation, explaining output
Output..
```

输出结果 0-1 一个输出结果样例

请注意上面这个输出结果中的细节——用户名（上面这个输出结果中的 user）（以及命令行提示符是\$还是#）能够告诉你，这条命令是在 shell 中就能执行，还是必须要有 root 权限才能执行的。主机名（上面这个输出结果中的 hostname）则可以告诉你这条命令是在哪台设备上执行的——如果它是 generic，表示是在一台模拟器中；如果是 flounder，表示是在一台 Nexus 9(L) 中；如果是其他移动设备的名称（s3、s4、kindle 或 Nexus 5 之类的），则表示是在一台对应的移动设备中；如果它是 Forge，则表示是在作者自己的 Linux 计算机上运行的。我尽量避免在书中出现大段的代码（至少在本书中是这样的），在迫不得已的情况下，我也尽量只给出最关键的代码，此外，我也会在代码中加上帮助你理解代码的注释。代码的字体颜色也调整为能够同时兼顾彩色（如果你读的是 PDF 版）和黑白（如果你阅读的是纸质版）两种打印方式的颜色。

## 最后……

本书绝对是个大工程，我像在大海里捞针那样把最重要的代码从 Android 源码里挑了出来。但即便是这样，肯定还有人想要亲自查看相关源码。所以，我在每次讨论中都会给出相关的源文件，并且是以超链接的方式给出的（纸质书的读者就只好抱歉了，我会把它们统一放在一个表格里供你们查阅）。有兴趣的读者也可以去谷歌的 Android 源码网站，或者去 <http://source.android.com/> 用 git 或者 repo 命令下载最新版的 Android 源码。

这本书是我“一个人的战争”——除了封面设计（封面设计是 Dino Tsiopanos 送给我的礼物，Dino 不仅是个很棒的工程师，还是个很棒的画家）。书中的所有内容，包括文字、图片、排版和编辑都是我一个人完成的。幸亏我还可以向我的两位审校 Moshe Kravchi 和 Arie Haenel 寻求帮助，我真的非常感谢你们二位！Nikolay Elenkov（那本棒极了的 *Android Security Internals: An In-Depth Guide to Android's Security Architecture*<sup>1</sup> 一书的作者）也对本书提出了宝贵的意见和建议。Aviv Greenberg 在出版前的最后一段时间里以最快的速度通览了全书，给了我非常重要的评论。我还要感谢 Eddie Cornejo，你不光挑出了很多错别字，还让我有底气说：我（在书中）对所有操作系统的评价都是公正的。最后，我还要感谢 Nikola Veljkovic，感谢你帮我修正了书中的许多打字错误。

此外，我还要特别感谢 Yoav Chernitz。事实上，我在所有书中都应该感谢你——因为正是在你的鼓励下，我才会走上写书这条道路。或许对于我来说，这一切都是不言自明的，但我还是要将这件事告诉本书的读者，补上在本书的第 1 版（更新至 Android Marshmallow 版之前的那一版）中，没有对你表示特别感谢的缺憾。出于同样的理由，我还要特别感谢 Yobo，因为你引领我走上了 Android 研究之路，是你告诉我 Technogeeks 有 *Linux to Android* 和 *Android Internals* 这两门课的培训需求——它们现在已经是最热门的两门课程了，而且也构成了这套书（共两本）的基础。另外，如果不是 Ronnie Federbush，我也不会考虑像现在这样，尝试自媒体出版，而且本书的第 2 本（即将出版）和 *Mac OS X and iOS Internals* 第 2 版也将以这一形式出版。

这里还要特别说一句，我个人最诚挚的感谢要送给 Amy，我俩在一起世界才完美，从我上一本书开始（实际上是在所有的事上）你一直给予我无限的支持和鼓励。这是我永远要唠叨，永远不会忘记的感谢！

本书是我用 vim 编辑器以符合 HTML5 的格式要求全手工输入的（对，我保证，我确实就

---

1 中文版为《Android 安全架构深究》，刘惠明、刘跃译，电子工业出版社，2016 年 3 月出版。——译者注

是这么干的！不过这次也是够了，我想下次我再也不会这么干了），所有的图片全是用 SVG（又让我幼小的心灵受到了一次严重的创伤！）或 PowerPoint 画的，这也就能够解释为什么这本书花了这么长的时间才面世。不过好消息是本书的第 2 本（大概比这一本厚一倍！）应该也会在不久后出版。在书中做索引也是一件极其令人痛苦的事，所以我决定：在更新内容时，就不再做索引了（你只要直接在 PDF 里搜索相关内容就行了）。如果你在书中发现了任何格式错误或者技术错误，也请体谅一下我这些辛苦的工作！不过对于技术错误，我提供专门的奖励——每发现一个，就会有一定的奖励进入你的腰包。

我还维护了本书的官方网站，在这个站点上还有更多的进阶阅读材料和一些专门定制开发的工具，网址是：<http://NewAndroidBook.com/>。本书的勘误表——包括错别字和错误修订（我希望不会有），也会放在这个网站上。

如果你想要在推特上找我，请关注我公司的官推@Technologeeks，那里经常会推送一些我的书的更新以及扩展阅读材料。Technologeeks 公司也提供关于 Android、OS X、iOS、Linux 及其他操作系统的专业咨询和培训服务，所以我也建议你关注一下我公司的网站 <http://Technologeeks.com/>。特别要说一句的是：在我公司提供的培训服务中，关于 Android 和 OS X/iOS 的培训都是基于我这两本书的。我公司在领英里还有一个 Android Kernel Developers 群——如果你有兴趣的话，也可以加入这个群，来打个招呼（或是提些问题）。

我衷心希望你能觉得这本书的内容既有意思，又吸引人（好吧，我想应该是在技术书籍中比较吸引人）。在本书的官网上我搭建了一个论坛，恭候您的批评和建议。

现在就让我们正式开始吧！



## 第 1 章

# Android 体系结构的变革之路

尽管 Android 是建构在 Linux 基础之上的，而且非常依赖于其中的许多基础设施 [其中最明显的就是内核 (kernel)]，但 Android 已经成为一种独一无二的操作系统。与相互间共享了 (除了 UI 和一些框架之外的) 大部分底层代码的 OS X 和 iOS 不同，Android 中引入了大量的框架以及支持这些框架的运行时 (Dalvik)。事实上，多数面对用户的特性和版本升级时所做的增强，都是以新增框架或添加 API 的形式完成的，只有很少一部分涉及内核级的代码。

本章讲述的是 Android 系统版本的历史变迁过程及 Android 的体系架构。我们从 Android 系统的版本历史变迁过程 (从 Cupcake 1.5 到 Marshmallow 6.0) 讲起，在讨论时会同时关注各个版本中系统相关的特性和系统版本升级时所做的增强。然后，我们将去讨论 Android 的体系架构，把它和 Linux 的体系架构放在一起做比较和比对。系统中的每一层我们都会详细讨论，以便为在本书之后的各个章节 (以及本书的第 2 本) 中对它们做更进一步的详细讨论时打下一个良好的基础。最后，我们的讨论还会涉及各种 Android 的衍生产品，以及在 Android 这个进化速度非常快的操作系统的下一个版本中，可能会出现的各种性能增强。



本书付印时，Android 5.0 (Lollipop) 已经可以安装在谷歌的 Nexi 手机上，并且计划首次向生产商展示。事实上，Android 的进化速度非常快，不论你是什么时候拿到本书的，当你读到这里时，Android 很可能又有了新的变化——由于移动操作系统之间激烈的“军备”竞赛，Android 每隔几个月就会出一个新的版本。为了尽量跟上这一节奏，本书经过两次修订 (2015 年 11 月修订，当时的最新版是 Marshmallow；2016 年 11 月修订，最新版为 Nougat)，目前已经更新到了 Nougat 7.1 版 (2016 年 11 月)，反映了在这之前 Android 的版本变迁情况，但恐怕还是跟不上 Android 飞一般的更新速度。所以，也恭请读者经常关注本书的官方网站 ([NewAndroidBook.com](http://NewAndroidBook.com)) 以随时获取最新的动态信息。



## 1.1 Android 系统版本的历史变迁

在 Android 问世短短 7 年的时间中，它已经更新了不下 12 个版本了。如果我们只看 Android API 的版本号 [版本号把 Android 内部使用的 API 集与 Android 各个版本（用食品名称命名）的产品代号挂上了钩] 的话，这个数字已经增长到 23 了。一一列举每个版本新引入的各种框架特性未免过于冗长，也许还会不可避免地漏掉一些。所以我把这一节的目标定义为从更加技术的角度，重点阐述这些不同版本的 API 在系统层面（而不是框架层面）及其他方面的一些明显的不同之处。如果想要更进一步了解各个版本之间的变化情况，我建议你阅读维基百科中的综述网页<sup>[1]</sup>，或者阅读各个对应 Android 版本的文档。

表 1-1 中给出了 Android 系统版本的历史变迁过程，并且给出了各个官方发布的 Android 版本与各个 API 版本及内核之间的关系。注意：即使使用的是同一个版本的 Android 系统，各种品牌、型号的手机的内核版本也不一定是一样的，因为有些厂商会编译他们自己的内核，或者把系统迁移到更新版本的内核上去。

表 1-1 Android 的历史版本（按时间排序）

发布年份	产品代号	Android 版本	API	内核版本	市场占有率
10/2016	Nougat	7.1	25	4.1	<0.1%
8/22/2016		7.0	24	3.14-3.18	<0.3%
10/2015	Marshmallow	6.0	23	3.4(armv7) 3.10(arm64)	24.0%
3/2015	Lollipop	5.1-5.1.1	22		34.1%
11/2014		5.0-5.0.2	21		
10/2013	KitKat	4.4-4.4.4	19/20	3.4	25.2%
07/2013	JellyBean(MR2)	4.3	18		13.7%
11/2012	JellyBean(MR1)	4.2-4.2.2	17		
07/2012	JellyBean	4.1-4.1.1	16	3.0.31	1.3%
12/2011	Ice Cream Sandwich(MR1)	4.0.3-4.0.4	15		
10/2011	Ice Cream Sandwich	4.0-4.0.2	14	3.0.1	
07/2011	Honeycomb(MR2)	3.2-3.2.6	13	2.6.36	< 0.1%
05/2011	Honeycomb(MR1)	3.1	12		
02/2011	Honeycomb	3.0	11		
02/2011	Gingerbread(MR1)	2.3.3-2.3.7	10	2.6.35	1.3%
12/2010	Gingerbread	2.3-2.3.2	9		

续表

发布年份	产品代号	Android 版本	API	内核版本	市场占有率
05/2010	Froyo	2.2-2.2.3	8	2.6.32	0.1%
10/2009	Éclair	2.0-2.01, 2.1	5-7	2.6.29	< 0.1%
09/2009	Donut	1.6	4	2.6.29	< 0.1%

实际使用数据是经过谷歌编辑的，这一数据可以在 Android 开发者网站<sup>[2]</sup>上通过导航面板找到。这也是表 1-1 中“市场占有率”一栏百分比的数据来源（更新日期 2016 年 11 月）。由于事实上已经不会有什么设备还在使用比 Froyo 版本更早的 Android 系统了，所以本书将不会去讨论那些系统。

### Froyo（冻酸奶）



Froyo（Frozen Yogurt）是第一个支持在扩展存储设备（也就是 SD 卡）上安装应用（App）的 Android 版本。为此，它专门引入了 ASEC（Android Secure Containers，Android 安全存储容器）的概念，以保护（通常是使用 FAT 文件系统的）扩展存储设备中安装的程序不会被复制出去（关于 ASEC 机制的讨论详见第 2 章）。这个版本中引入的另一个非常有用的特性是 USB tethering（USB 网络共享技术，也就是电脑和手机用 USB 线连上之后，电脑就可以把手机当成网卡，通过手机上网）。最后，Froyo 还通过引入即时编译技术（JIT，Just-In-Time compilation，Froyo 中用一个专门的线程来完成这一工作），在很大程度上提高了 Dalvik 的运行速度。

### Gingerbread（姜饼人）



Gingerbread 是第一个被广泛采用的 Android 版本，而且理由十分充分：它引入了许多特性，对系统做出了重大的改进。在 Dalvik 层面上，引入了并发的垃圾回收机制，使 Android 能在运行应用的同时，运行 GC（garbage collection，垃圾回收），而不必像过去那样，在运行 GC 的过程中要把应用暂停下来，这极大地提升了应用的响应速度。这一改进的意义类似已经在 Froyo 版中引入的即时编译（JIT）机制。与传感器相关的 API 也经历了完全的更新，与传感器相关的 HAL 层（硬件抽象层）被大大地扩展，以支持更多类型的传感器，同时也提高了对原生代码的访问性。对原生代码支持度的提升也体现在其他方面：在这一版本中，原生代码可以访问声卡、显卡、存储设备甚至可以访问 activity manager。Gingerbread 也首次引入了对 NFC（Near-Field-Communications，近场通信）的支持——尽管要到很久以后（随着冰激凌三明治版

的普及) NFC 装置才能被各大 Android 设备制造商广泛采用。

另一个值得注意的改进是: 支持 OBB——opaque binary blobs(也被称为“APK 扩展文件”)。Android 中应用 apk 包的大小不能超过 50MB, OBB 是解决这一限制的变通方案, 同时它还能让应用对存放在其中的数据进行加密, 关于 OBB 文件的进一步讨论详见第 2 章。最后, Gingerbread 采用 Ext4 代替 YAFFS, 作为默认的文件系统。

除了上述这些改进之外, Gingerbread 最被诟病的一点是: 它是迄今为止最不安全的 Android 版本。除了系统自带的短信应用里存在 bug 之外(它会把短消息发送到错误的接收方那里), 它还被曝出了许多安全漏洞——这些安全漏洞导致移动端 rootkit 级的恶意软件爆炸式地涌现出来。

## Honeycomb (蜂巢)



Honeycomb 把 Android 带入了平板电脑的世界中, 事实上这是个“平板电脑专用”的 Android 版本, 因为这个版本的源码树从来没有被完整地发布过, 官方也从来没宣布过它可以用在手机上(尽管如此, 有些厂商仍然试过在手机上使用它)。这一版主要的改变在于引入了 fragment 机制——它类似于 Windows 里的 MDI (Multiple Document Interface, 多重文档接口), 使得多个客户区 (client area) 能够同时共存, 而不用再像过去那样只能使用单一布局架构 (single layout architecture)。

Honeycomb 在图形显示方面也做出了重大改进——引入了硬件加速的 OpenGL 渲染 2D, 此外还引入了 Renderscript——这一 Android 自己的类 GL 语言。

另一个重要的特性是: 这个版本把 Android 带入了存储加密 (storage encryption) 的时代。Honeycomb 是第一个支持在底层加密用户数据分区的 Android 版本, 这把 Android 又带回到了与 iOS 4 (iOS 在这个版本中也引入了这一功能) 同一个起跑线上。在 Android 中, 磁盘加密是由 Linux 的 device mapper 来完成的。这被视为继 Froyo 中引入 ASEC 之后, 又一个重大进步。

比起 (加密) 用户空间这一特性, 在 Android 中引入多核支持这一点更为重要。这主要涉及要重新编译 Linux 内核以支持 SMP<sup>1</sup> (系统是否支持 SMP 可以用 Busybox 中的 uname 工具, 或者查看 /proc/version 伪文件的内容得知)。平板电脑是第一种使用多核处理器的 Android 设备, 不过现在, 除了最便宜的设备之外, 多核处理器已经被用在所有的 Android 设备上了。Android 的文档<sup>[6]</sup>中详细描述了如何修改代码才能让系统变成“SMP 安全的”<sup>2</sup>——大多数的修改集中在

1 SMP 是“Symmetric Multi-Processor” (对称式多处理器) 的缩写。——译者注

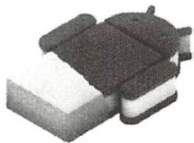
2 SMP Safe, 指软件可以在 SMP 结构的计算机上正常运行。——译者注



原生代码上，不过 Java 代码也要做部分调整。

Honeycomb 也是唯一一个源码（除了一些精心选择的部分之外）没有被公开的 Android 版本。这引起了部分厂商的恐慌，并使他们认识到这样一个事实：尽管 Android 是免费的，但是谷歌仍然控制着该系统。只要谷歌愿意，在将来的某个时刻，它完全可以修改 Android 的许可协议。

## Ice Cream Sandwich（冰激凌三明治）



冰激凌三明治（ICS, Ice Cream Sandwich）版给 Android 带来了许多变化，仅从它的版本号 4.0 就可以看出这一点。除了对 UI 做了大量的加强之外，用户能够明显感受到的变化还包括网络连接方面的众多新特性——Android 的 VPN 框架、Wi-Fi Direct 和 Android Beam。

ICS 中新增的另一个 API——`onTrimMemory()`回调函数，往往会被开发者们忽视掉。这个 API 会在系统内存不足时被调用，根据传进来的 `level` 参数所指定的内存不足的等级，应用应该释放尽可能多的内存，以避免被系统杀掉<sup>1</sup>。不过，请注意，这个 API 只是建议开发者使用的一应用也可以选择完全无视这个回调函数（而且事实上，有很多应用也是这样做的）。

## JellyBean（果冻豆）



对于用户来说，JellyBean 最突出的特性是：它支持在同一台设备上有多个用户账户。这一特性更多地被用在平板电脑而不是手机上（并且在官方文档中，只能在前者中启用），它使多个用户能够操作同一台设备——尽管能够活跃登录的（actively logged on）用户只有一个，但每个用户都有不同的 UI——有完全独立的 widget、应用，而且最重要的是，应用的数据也是独立的。我们将在第 8 章中详细讨论这一特性的实现方式。

除了多用户和其他一系列的 UI 特性之外，JellyBean 在 Froyo 版中开始提供的 ASEC 容器的基础上，还提供了应用加密和“预先锁定”（forward lock）的特性。那时，Android 的开放性带来的缺点之一是存在一个很容易盗版应用的方式：把应用安装到 SD 卡中去，然后在各个移动设备间“分享”。ASEC 提供了一个安全存放数据的容器，应用可以加密存放在其中的数据，并使之只能被使用该应用对应 uid 的进程读取（尽管在已经被 root 了的设备里还是没什么用）。我们将在第 2 章中讨论这一特性。

1 关于内存不足时，系统的应对方案，详见第 7 章“内存不足时的应对方案”一节。——译者注



JellyBean 对应有三个 Android API 版本，这三版本间自然也引入了许多新的变化，下面我依次介绍一下：API 17 首次在 Android 中引入了 SELinux（详见第 8 章的讨论），并且修复了使用 ADB 进行 USB 调试前，强制授权认证过程中的一个漏洞。API 18 引人注目的改变是支持 OpenGL ES 3.0，蓝牙 AVRCP（Audio-Video Remote Control Profile，音频-视频远程控制配置文件）1.3 和蓝牙低功耗（LE，Low Energy），以及可以用来调整应用权限的 App Ops 服务（不过这个服务的 UI 在稍后的 4.4.1 版中被移除了）。

## KitKat（奇巧）



Android 4.4 的产品代号是“KitKat”[而且这个名称实际上是源自谷歌和“好时”（Hershey）公司之间的商业合作伙伴关系]。这次谷歌是真的想要整合一把碎片化的 Android 世界：尽管 JellyBean 是最为流行的版本，但有很大比例的设备仍然在使用各个旧版本的 Android——需要注意的是，GingerBread 及更早的版本不光是已经被厂商放弃了，而且因为其 API 版本实在是太老了，新一点的应用甚至都没法在这些系统上运行。另外，中高端市场已经开始慢慢变得饱和了，而低端市场中 Android 又面临着 FireFox OS 及其他系统的竞争。

KitKat 最被看重的一点就是“瘦身”——首次提出要能在任何一台设备上（甚至是只有 512MB 内存的入门级设备上）提供流畅的操作体验。这样做的一部分动机是：通过提供一个所需资源更少，但操作更流畅的操作系统，能促使所有的厂商在所有的设备上（甚至是最低端的设备上）都用上最新版的 Android 系统，从而一举终结 Android 世界的碎片化状况。为了达成这一目标，需要做大量“水面之下”的修改，比如重写框架的代码、降低其所需的内存、以串行的方式顺序启动各个服务（以减轻内存压力）。为此还特意引入了一个能检测设备内存是否较少（low RAM device）的新的 API（`ActivityManager.isLowRamDevice()`，这个 API 会返回系统属性 `ro.config.low_ram` 的值）。使用这个 API，开发者可以检测出可用内存的数量，并据此对资源的使用做出相应的规划。KitKat 中还新增了 `procstats` 服务，尽可能地向开发者提供有关他们的应用使用内存的情况的信息。

对于那些内存确实有富余的设备，KitKat 则使用了 Linux 内核中另一个名为 zRAM 的新特性。事实上，这个特性甚至比 KitKat 本身还要新，直到 3.14 版（KitKat 用的是 3.4）才被官方认为是稳定的，并被整合进内核中，而谷歌则是在更早的时候就在 ChromeOS 和 Android 中采用了它。这一特性使得系统可以将一些当前未使用的内存页交换到内存指定的位置上去，以此解决移动设备的一个主要缺陷——乍一看，使用闪存作为存储设备的设备没有可以用来进行页交换的分区。把内存作为页交换的存储空间听上去是那么不科学，不过事实上这可是个聪明绝顶的改进——被交换出去的页是要先经过压缩的（这样就能腾出一定的内存空间来），而且这样

也有利于快速地把之前交换出去的内存页再交换回来。在使用了 zRAM 的设备上会有一个特殊的块设备（即/dev/block/zram0），它也就是/proc/swaps 文件。

这种“压缩内存”的方法也是在 iOS 7.0 中首次出现的，时间恰恰是在 KitKat 发布几个月前。KitKat 中另有几个有意思的特性可能也是跟风 iOS 7.0 的，其中包括计步器（软件定义的传感器，像是对苹果的 M7 协处理器的回应）以及计时器聚合（timer coalescing）和 sensor batching。后两个是意义重大的改进，它有助于延长电池寿命。为了做到这一点，Android 实际上是加大了计时的粒度并延缓了传感器更新的频率，使之间隔时间更长，且步调更趋一致。这可以在很大程度上增加电池的可用时间——既有直接的作用（CPU 空闲的时间段更长了），又有间接的影响（降低了设备被唤醒的总次数——每唤醒一次设备都会在用电量和性能方面产生一定的开销）。

KitKat 中其他一些值得注意的特性包括支持蓝牙 MAP、Infrared Blaster (ConsumerIr) API、新的打印（printing）框架以及 NFC HCE（host card emulation）。不过或许意义最深远的探索性改变还依然“犹抱琵琶半遮面”：引入了 ART（Android RunTime，Android 运行时），作为 Dalvik 虚拟机的一个可选的替代方案。

截至本书编写时，KitKat 已经经历四次版本更新了，目前它的最新版是 4.4.4。此版本更新的主要内容是：修复了一些 bug 并对摄像头功能做了增强，但并没有修改 API 的版本——尽管内部的 API 已经有了一些修改。KitKat 还是当前最常见的 Android 系统，（截至 2015 年 10 月）约 4 成的移动设备上装的都是这个系统。

## Lollipop（棒棒糖）



截至本书编写时，最新版的 Android 系统是 Android Lollipop。这个版本在用户界面上最显著的改变是引入了“Material Design”——一种扁平化的界面接口，其设计目标是为了让界面上的各种元素能够拥有真实的光影效果，并能在移动时保持这种效果。这种类似打印纸张的设计实在是让人不禁想起 iOS 的 UI 设计。这一版 Android 的另一个重头戏是它的通知系统，谷歌对通知系统的支持做了极大的扩充。

上述改变还只是冰山浮在水面上的一角，水面之下还有意义更大的改进：首先，也是最重要的是采用了 ART 虚拟机（Android Runtime）——它通过使用 AOT（Ahead Of Time，预先编译技术）而不是 JIT（Just In Time，即时编译技术）提前将 Dalvik 字节码编译成原生代码（Native code）的方式，为系统带来了很大的性能提升。除了带来性能提升之外，ART 还让 Android APP 能够充分利用 64 位处理器，这一点我们将在第 2 本中详细讨论。图形栈也得到了升级，以支持 OpenGL ES 3.1。音频系统也得到了改进，特别是能够更好地处理音频输入。类似地，摄像头

(camera) 相关的 API 也得到了完善。对传感器的支持（通过硬件抽象层）也有进一步的提升，使之能够支持更加复杂的手势，甚至还添加了对心率传感器的支持。这一版中的“明星项目”是“Project Volta”，这一项目的目标是（通过新的作业调度 API）增加电池寿命并提供更好的电源监视工具（尤其是 batterystats 服务）。最后，Lollipop 还为新的 Android TV 打下了基础。

Lollipop 的发布拖了相当长的时间，甚至可以说是有些痛苦的——从发布（2014 年 6 月）到官方发行（2014 年 11 月），这中间谷歌花了近 6 个月的时间，而被采用则是更加靠后的事了——几乎过了整整一年之后，每 4 台 Android 设备上安装 Lollipop 的也不足 1 台。在早期发布的版本中出现过重大的 bug（而且讽刺的是，这些 bug 竟然是和电源管理及性能有关的），这使得当时正全力部署 Lollipop 的厂商非常恼火——这一压力迫使谷歌相当迅速地将 Lollipop 升级为 5.1（对应的 API level 是 22）（2015 年 3 月），并推出了大量的补丁。Android 5.1（又称 Lollipop MR1）中增加了许多 UI Tweak，以及（更为重要的）一些值得注意的特性，包括 HD-Voice calling、双 SIM 支持、设备保护（这需要“kill switch”锁住被盗的手机并防止它被重置为出厂设置）。2015 年 8 月，爆出了两个影响当时所有 Android 设备（包括 5.1 在内）的重大漏洞（详见第 8 章的讨论）。

## Marshmallow（棉花糖）



Android Marshmallow 是本书第一版发布时，谷歌公司的最新版 Android 系统——它是谷歌公司于 2015 年 5 月 2 日发布的，但最终确定其产品代号则是近 2 个月以后的事了。在吸取了 L 版发布时的教训后，谷歌制定了非常严格的时间线，以控制三个开发者预览版的发布时间——在这样严格的措施下，尽管还是稍有延误，但这一版 Android 系统还是在 2015 年 10 月最终发布了。谷歌同时提供了模拟器中运行的镜像和（用于谷歌 Nexus 手机的）出厂配置的系统镜像文件，其中包括首次发布的、运行在 ARM64 位处理器上的系统镜像。因为那时，Android SDK 中使用的 QEMU 模拟器已经支持 ARM64 位处理器了。同时近乎完整的源码也能在 Android 的 Git 代码仓库中下载到。

从新增特性的角度讲，M 版再不是常规升级，而是一次近乎革命性的更新。尽管它所带来的一些值得注意的特性似乎是直接从 iOS 那里搬来的——包括支持移动支付、内置指纹认证授权（这一类似苹果设备里的 TouchID 的特性在 Lollipop 版中就已经引入了，但直到 M 版中用户才能在 App 中使用它）以及一个名为“floating toolbar”的在用户选中文字时会自动弹出的操作菜单的新特性。M 版中还新增了一个名为 gatekeeper 的服务，其通过使用完全由硬件实现的口令和秘密信息存储器，提高了 Android 这方面的安全性。



M 版带来的一个重大改进是全新的应用权限模型。在这一模型中，权限的强制检查和授予被移到了运行时(runtime)完成，而不再像以前那样，是在应用安装时执行的。这又让 Android 站到了和 iOS 一样的起跑线上——现在可以在敏感操作发生时提示用户，而不像以前那样，只能在应用安装时跳出一长串权限授予清单（详见第 8 章的讨论）。同时这也大大降低了恶意软件伪装成“愤怒的小鸟”之类的游戏，装进你的手机，然后偷偷窃取你的个人敏感信息或使用摄像头的风险。

M 版的另一个目标是改进之前版本中的两个不足之处：数据加密（数据加密是在 HoneyComb 版中引入，并在 Lollipop 版中默认启用的）现在可以扩展到对整个存储分区（甚至是扩展存储设备）进行加密了；电源管理（这是一个极具挑战性的课题）进一步改进为“Doze”模式，如果设备长时间进入休眠模式，设备在休眠期间会周期性地唤醒应用，以同步数据或完成其他被挂起的操作。M 版中还引入了应用空闲检测（App idle detection）（它有点类似于 iOS 中的“App Nap”特性），这一特性会挂起没在使用的应用。

其他一些更为独特的特性包括直接分享（Direct Share）、App Linking、改进过的音频/视频同步（包括快放/慢动作手势回放）、支持 MIDI、支持直接设置闪光灯（手电筒）等。另外，扩展了摄像头相关的 API，并对通知进行了改进，以及对“Android for Work”做了不少重大增强。完整的变动列表，详见 Android 开发者网站（<http://developer.android.com/about/versions/marshmallow/android-6.0-changes.html>）。

如果谷歌公司真的能够认真对待它们自己公布的时间表，M 版完全可能彻底碾压 Lollipop（按最乐观的估计，安装 Lollipop 的设备的百分比也不过徘徊在百分之十几之间）。厂商们完全可以选择稍微再等上那么一小段时间，而不是花上一段挺长的时间去升级到 Lollipop——只要在不远的将来，等 Marshmallow 正式发布之后，一次性升级到位就行了。

## Nougat（牛轧糖）



本书中文版的翻译过程中，谷歌于 2016 年 3 月初，宣布又出了一版新的 Android 系统，给人带来了意外的惊喜。这一举措有些打破常规，因为谷歌通常是在年末的谷歌 I/O 大会上发布它的新产品的。这一版 Android 系统最终于 2016 年 8 月 22 日发布，不过只能用在 Nexus 6 手机和 Nexus 9 平板电脑这两款设备上。

Nougat 带来了多窗口（multi-window）的功能（正如本书之前猜测过的那样），这一功能之前已经出现在 iOS 9（和一些三星设备）上了，实在是凑巧的很啊，哈哈。这一版本中，还支持“重按”（force-touch）手势——毫不意外，这也是为了紧紧跟上 iOS 上的“3 Touch”。其



他一些改进，比如绑定通知（bundle notification）和对（正在不断改进中的）Doze 进行的改进，也已经发布了。同时发布的还有名为“Vulkan”的新的 3D 渲染 API 的支持，并在平台上支持 VR 应用。此外还有一些重要的安全改进，其中比较主要的有：使用 seccomp-buf 只让有权限的应用才能使用某些系统调用，限制（第三方应用对）ioctl(2)的访问，以及在设备启动过程中验证/system 分区有没有被修改过。在（虚拟机中使用的）处理器体系结构方面，由于谷歌转而使用 Oracle 的 OpenJDK，开始支持 Java 8 的特性（比如闭包）并使用新的 Jack 编译器，使得 Dalvik 虚拟机中使用的字节码也有一些改动。此外，在 Nougat 版中还引入了一种新的启动模式——“direct-boot”，在这种模式下，一些应用可以在系统启动之后就能开始运行，而不必像之前那样，要等到用户解锁屏幕之后才会运行。

Nougat 发布后仅仅不到两个月，谷歌就宣布了 Nougat 7.1，把 API 的版本升级到了 25，并增加了一些加固安全的特性——其中最值得注意的是基于文件的加密（紧追 iOS 的 cprotect 机制），我们将在本书的稍后部分中详细讨论它。

### 实验：查看你的设备的 Android 版本

尽管厂商会以各种方式定制 Android 系统，但是底层系统还是一样的。在大多数 Android 用户熟悉的通过“设置”→“系统”→“关于手机”等操作步骤打开的图形界面中，就能提供关于当前使用的 Android 系统版本号的详细信息。与之对应的类是使用 android.os.Build 类的 com.android.settings.DeviceInfoSettings 类（可以在 AOSP 的 packages/apps/settings 中找到它），不过界面上显示出来的相关值是从一些系统属性中读取出来的。因此，获取这些值的更简单的方式通常是直接使用 getprop 这个工具。相关界面及对应的具体属性如图 1-1 所示。

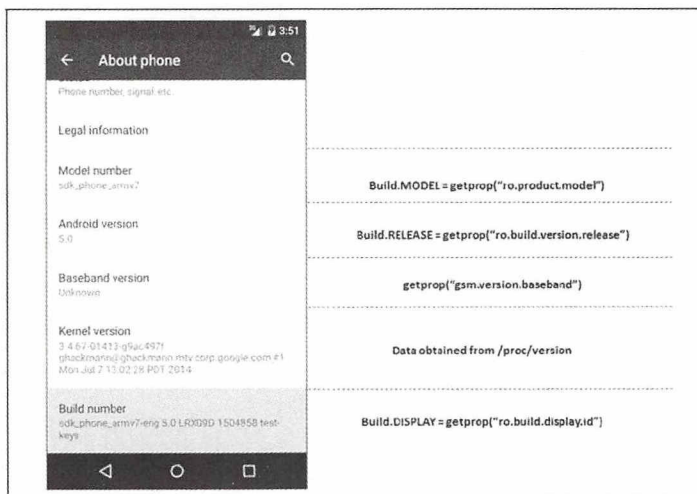


图 1-1 Setting App 中 DeviceInfoSettings 类展示的信息与各个系统属性间的关系

系统属性（见表 1-2）中记录的值（这些信息是由 AOSP 产生并记录在/system/build.prop 中的）在那些被修改过的系统中，仍会保持原来的值——即使是在被 Amazon 公司重度定制过的“FireOS”系统中也是如此。最有用的属性当属 ro.build.version.sdk（API 版本号）和 ro.build.fingerprint。对于 ro.build.fingerprint 这个属性，其中斜杠隔开的就是另外几个其他属性的值，例如某个 ro.build.fingerprint 的值是：

```
generic/sdk_phone_armv7/generic:5.0/LRX09D/1504858:eng/test-keys
```

表 1-2 系统属性

属 性	描 述
ro.product.manufacturer	厂商的 ID
ro.product.name	设备代码名称（Device Code Name），如果是谷歌的设备，那么这里是各种鱼类的名称
ro.build.product:version.release	产品名称及 Android 基本版本号
ro.build.id	首字母：系统版本号（其余部分详见参考文献[11]文档的描述）
ro.build.version.incremental	内部 build 的版本号，它的值是 AOSP 的 build 系统自动增加的
ro.build.type	user：表示是正式发布供用户使用的版本（User Facing 的缩写）；eng：表示的是工程测试版/内部版本（Engineers 的缩写）
ro.build.tags	release-keys：正式发布的系统，使用的是真实的证书；test-keys：开发测试的系统

## 1.2 Android 与 Linux

### 并非另一个 Linux 发布版本

Linux——Android 的核心部件，甚至在 Android 这个构想提出来之前，就已经好好地存在了几十年了。Linux 是一个完全开源的操作系统。Linux 的内核起源于 Linus Torvalds 的硕士论文，之后受到了全世界的赞誉并被广泛使用。当然，仅仅就只有一个内核本身，并不能构成一个完整的操作系统。Torvalds 因此决定把他的工作以开源的形式共享出来，以吸引更多的开发者进一步扩展它（以为它提供组件的方式），这些二进制可执行文件形式提供的组件既有从其他 UNIX 系统移植过来的，也有专门为 Linux 开发的。当时 UNIX 操作系统的售价都是很贵的，而 Linux 作为一种免费的 UNIX 系统，马上就爆炸式地流行了起来，并成功使得其他 UNIX 系统售价大幅降低，最终导致大部分 UNIX 系统消亡。

随着它的飞速发展，Linux 也引起了商业公司的兴趣。一时间，大量的公司（这些公司的

唯一目标就是把 Linux 内核和一些其他二进制可执行文件打包在一起，然后发布出来）纷纷涌现，并提供了各种不同的 Linux “发布版本”。这些公司通常都会根据他们所支持的完整商业模式，免费提供 Linux 系统。但有时，也会提供含有定制工具或专用工具的“专业版”或“企业版”等进阶版系统——当然这些进阶版的系统是需要花钱购买使用许可的。

事实上，Linux 也迅速成为嵌入式系统的操作系统。相对于这一领域中的其他竞争者——比如 Windows CE（它太耗资源），或者 PSOS 或 VxWorks 这类实时操作系统（使用这两种操作系统都要花费重金购买使用许可），Linux 提供的平台不仅免费，而且还是轻量级的，并且完全可以定制。MontaVista 公司根据它自己完整的商业模式，将 Linux 移植到了嵌入式系统中——特别值得注意的是：它支持 ARM、MIPS 以及 PowerPC 等体系结构的处理器。移植到嵌入式平台上的 Linux 系统提供了与桌面版的 Linux 系统一样的功能——一个提供完整特性的 shell 环境。而购买这一切使用许可的费用也不贵。

但是开发者的胃口更大。shell 接口的时代已经一去不复返了，所有的用户（那些在命令行中经历了千锤百炼的骨灰级老鸟除外）都希望他们的操作系统有一个图形化的用户界面。Linux 使用 X-Windows（传统 UNIX 系统中使用的窗口架构）作为它的 GUI 界面。但是在嵌入式系统中构建一套 GUI 又谈何容易！再说用 X-Windows 的 API 编写图形界面程序也是件极为痛苦的事。另外，MontaVista 之类的厂商提供的也只是基本的平台，开发者仍然需要把额外的组件移植过来，或者自行编写这些组件——常常不得不从零开始。

## 然后 Android 就登场了

2005 年，谷歌承诺会开发一款移动操作系统，当时他们收购了 Android——Andy Rubin 创办的一家小公司。Android 在消失了一段时间后，于几年后回归（就在苹果公司发布“iPhoneOS”后不久）。当时正力图适应发生着革命性变化的设备的移动生产商们，希望马上就能拿出一款拥有类似 iPhone 的用户体验的产品，而且要能让他们快速上手。

由于所提供的东西正合厂商的心意，Android 获得了戏剧性的成功——它并不是另一个 Linux 发行版本，而是一个完整的“软件栈”（software stack）。术语“栈”意指多个层（layer）。Android 提供的不光只是一个基本的内核和一堆能在 shell 中运行的二进制可执行文件，它还自带一个完整的 GUI 环境以及丰富多样的框架（外加便于使用的开发语言 Java），Android 向开发者提供了一个真正的快速应用开发（RAD，Rapid Application Development）环境，这样开发者就能直接调用框架中预先写好并经过良好测试的代码，只需寥寥数行代码就能使用诸如摄像头、运动传感器、GUI Widgets 等各种高级功能。Android 从 iOS 那里“借鉴”了众多的特性，并持续不断地改进它们，这一切最终使 Android 成为事实上的移动设备标准操作系统——就像



Windows 之于台式计算机一样。

Android 也经常不断地从它自己的生态系统 [Android 中的“App 市场”（这是迅速跟进苹果公司的“App Store”的结果）] 那里获得反馈并进行改进，通过这一正循环，巩固其市场占有率，采用这一模型使得开发者们能够以一种远为（甚至会有人说是“极其”）轻松且基本不存在障碍的方式，快速发布他们的 App。其结果就是“谷歌 Play 市场”（这是“App 市场”现在的称呼）已经胜过了 App Store，并能提供数以百万计的 App。采用 Android 系统让移动设备制造商能够立即访问和兼容这些 App——只要他们签署一下谷歌的移动应用发布协议（MADA，Mobile Application Distribution Agreement）就可以了，这个协议授权厂商能够不受限制地整合谷歌的 App 和服务。

从某种意义上说，Android 对 MontaVista 及其他嵌入式 Linux 公司的所作所为，就相当于当年 Linux 对 UNIX 及其他竞争对手所做的那样——通过提供一个完全免费的替代品，对市场进行了一轮“收割”。谷歌推出免费的 Android，完全不用支付任何许可费用（至少到目前为止不用），使用条款也相对宽松（尽管现在这方面的政策也在缓慢而坚定地收紧）。毫无疑问，Android 在短短的几年里取得了巨大的成功，以全球移动市场约 80% 的占有率，实现了无可置疑的霸权，目前只剩下了唯一一个对抗堡垒 iOS（目前大约拥有 20% 的市场占有率），以及几乎可以忽略不计的 Windows Mobile 和黑莓。在选择操作系统的问题上，移动设备制造商基本就没几个选项：要么自己另搞一个，要么使用一个已经做好的系统——几乎所有厂商的选择都是后者<sup>1</sup>，而他们的选择范围也无外乎 Android 或是 Windows Mobile。微软试图跟上 Android 的节奏，免费提供它自己的系统，但是这一努力来得太晚，也太迟了，因而它也缺乏自己的生态系统。黑莓则有自己的打算：它已经把 Android 的运行时移植到了它自己的操作系统中，希望通过提供兼容大量 Android App 的运行环境的方式，重新赢回市场。

## 与 Linux 的异同

Android 是架构在 Linux 之上的，但也对它做了大量的修改——其中包括部分地打破了 Linux 主流版本间的兼容性。Android 内核源码树大概是在 2.6.27 版本中从 Linux 内核主线上分离出来

---

1 移动设备生产商也越来越担心 Android 的几个弱点：首先是共用的基础特性，这使得他们产品的同质化竞争越来越厉害。其次，他们也越来越依赖于谷歌公司，而谷歌事实上一直致力于规范和统一各种设备中 Android 系统的外观和用户体验。最后，谷歌的移动应用发布协议（MADA）强制要求在设备上安装谷歌公司的所有 App，才能访问“谷歌市场”。这些问题使得一些厂商（特别值得注意的是三星）开始寻找 Android 的替代品（比如 Tizen）。不过目前来看，Android 的地位看上去还是稳固得很，在可以预见的一段时间里不太可能丧失市场占有率的优势。——原注



的，但到了 Android 3.3 又开始回归 Linux 内核主线了。在用户模式上，谷歌在一个独立的完整代码仓库中，维护着各种框架（framework）和 AOSP（Android 开源项目，Android Open Source Project）的运行。从高级视角出发，尽管很难非常准确地说出这两个操作系统之间到底有多少不同，但是大致的估算 Android 和 Linux 在内核层上有 95% 的相似之处，而在用户模式下有约 65% 的相似性。

这一估算的依据是：考虑到在内核级别上，除了少数一些区别之外（ARM 处理器及一些硬件的驱动是非加不可的），内核源码的剩余部分并没有被修改过。这些不同（包括 IPC、内存以及日志增强）归在一起，统称为“Android 特有的特性”（Androidisms）。不过到目前为止，其中的大多数事实上已经被合并到 Linux 主线上去了。有些替换了内核中类似的功能，有些被放在了 `drivers/staging/android` 目录中。

而在用户态这一级上，由于引入了两个全新的组件 Dalvik 虚拟机运行时和硬件抽象层（Hardware Abstraction Layer），再加上替换了 Bionic 的 `glibc`，以及提供了一个定制版本的 `init`（系统启动守护进程），Android 和 Linux 的分歧就大多了。不过尽管是这样，操作系统更多底层的大部分组件仍然没有被修改过，那些原生二进制可执行文件以及进程和线程的行为仍然和它们在 Linux 系统中时一模一样。这使得我们在本书中可以以此为切入点进行讨论，比如在第 7 章中，我们可以讨论基于 Linux 的底层调试和跟踪（trace）技法。

Android 还更巧妙地使用了一些 Linux 中已有的特性——这些特性大多存在于 Linux 桌面发布版本中，但往往还没有被使用。这些特性包括控制组（control group）、内存不足时的应对方案（Linux 中也有 OOM，但 Android 把它扩展成了它自己的内存不足时的进程清理器）以及一些安全特性——权能和 SELinux（详见第 8 章讨论）。

Android 中也使用了一些开源项目，这些项目在 Linux 中并不怎么流行，但却是支持 Android 的特性的骨干柱石。这些项目（位于 AOSP 的 `external/` 目录中）主要是负责实现 Android 的网络功能的，包括 `racoon`（vpn）、`mdns`（服务发现和 Wi-Fi Direct）、`dnsmasq`、`hostapd`（tethering 和 Wi-Fi Direct）和 `wpa_supplicant`（Wi-Fi）。还有一些其他的开源项目提供了库一级的支持（我们将在表 1-3 中讨论和展示它们）。

图 1-2 中是 Linux 和 Android 软件栈间差异的对比图。接下来我们会依次讨论这些应该引起我们注意的差别。

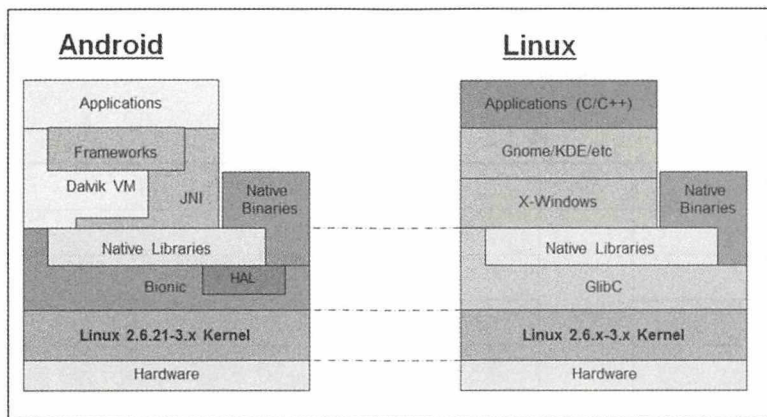


图 1-2 Android 与主流 Linux 体系结构间的比较



大多数开发者可能熟知谷歌提供的 Android 官方架构图（你直接搜索“Android Architecture”就能找到它），那张图实在是令人作呕（事实上，其他关于 Android 的书籍中，都会在简介里附上这张图，而且通常连颜色都是一模一样的）。那张图的作者出于自己的私心对图进行了简化，而且并没有准确地表示出各个层（layer）的位置关系（比如 JNI 就完全被无视了）。当各个层展开后，谷歌的旧图和我给出的图之间在架构关系表示上的细微差别有可能会“造成一些问题”（如果不是更严重的话）。好在本书付印时，谷歌终于在 Android 源码网站上<sup>[12]</sup>又给出了一张更为准确的（视觉上也有吸引力的）示意图。

## Android 的框架

Android 取得成功的关键因素之一就是它丰富的框架集。没有这些框架，Android 可能会和其他一些嵌入式 Linux 发布版本一样混得很差（或者也可能会步 MontaVista 的后尘——MontaVista 是一个在 Android 正式登场前，曾经一度非常流行的操作系统）。通过提供各种框架，Android 让应用可以很方便地创建进程，允许开发者使用高级的 Java 语言而不是底层的 C/C++ 语言进行编程。各种框架的不断增加也在进一步强化这一过程，因为有大量的用于进行图形、音频和硬件访问的 API 可供开发者使用。这一点与 X-Windows 和 GNOME/KDE 不同——这些系统简洁得多，操作也是以更直接的方式进行的。

使用 Java 的包命名规则后，Android 的框架会根据它们各自不同的功能被分割在各自不同的命名空间（namespace）中。位于命名空间 `android.*` 中的包是可以供开发者使用的，而位于 `com.android.*` 中的包则是仅供系统内部使用的。Android 也支持大多数位于命名空间 `java.*` 中的标准 Java 运行时包。表 1-3 对常用框架包（按相关框架被引入 Android 的 API 版本号排序）做了总结，这样就能让你对操作系统特性的演化有个大致的印象。注意：表中仅仅给出了各个框



架首次出现于哪个 API 中，并没有给出这些框架的后续扩展情况。同一个框架可能会在多个 API 中都有所扩充，因为系统在不断地给它新增更多的类。

表 1-3 Android 中使用的框架

包 名	API	作 用
android.app	1	支持应用的各类常规操作
android.content		提供使用 Content 的各种方法
android.database		支持数据库——主要是 SQLite
android.graphics		图形支持
android.opengl		支持 OpenGL 图形功能
android.hardware		支持摄像头、输入以及传感器
android.location		支持地理位置定位
android.media		支持多媒体
android.net		支持网络操作，java.net 中的各个 API 都是构建在其上的
android.os		支持操作系统的核心（core）服务和 IPC
android.provider		Android 内置的 content-provider
android.sax		SAX XML 解析器
android.telephony		支持电话的核心功能
android.text		文字渲染
android.view		UI 组件（类似于 iOS 中的 UIView）
android.webkit		Webkit 浏览器控制组件
android.widget		应用的窗口小部件（widget）
android.speech	3	提供语音识别和“语音”到“文字”功能的组件
android.accounts	4	支持账号管理和身份认证
android.gesture		支持定制的手势操作
android.accounts	5	支持用户账户
android.bluetooth		支持蓝牙
android.media.audiofx	9(G)	支持声音特效
android.net.sip		使用 SIP（会话初始协议，Session Initiation Protocol）（RFC3261）支持 VOIP
android.os.storage		支持 Opaque Binary Blobs（OBB）
android.nfc		支持 NFC（近场通信，Near Field Communication）



续表

包 名	API	作 用
android.animation	11(H)	view 和 object 的动画
android.drm		数字版权管理 (DRM) 和版权保护
android.renderscript		RenderScript (一种类似 OpenCL 的计算机语言)
android.hardware.usb	12	支持 USB 外围设备
android.mtp		支持以 MTP/PTP 方式连接摄像头等
android.net.rtp		支持实时传输协议 (Real-Time-Protocol) (RFC3501 <sup>1</sup> )
android.media.effect	14(I)	支持图像和视频特效
android.net.wifi.p2p		支持 Wi-Fi 直连 (Wi-Fi Direct) (点对点)
android.security		支持 keychain 和 keystore
android.net.nsd	16(J)	基于多播 DNS (Multicast DNS) 的邻接节点服务发现协议 (Neighbor-Service-Discovery) (Bonjour)
android.hardware.input		输入设备监听器
android.hardware.display	17	支持外接或虚拟显示器
android.service.dreams		支持 Dream (屏保)
android.graphics.pdf	19(K)	PDF Rendering
android.print[.pdf]		支持外接打印机
android.app.job	21 (L)	任务调度
android.bluetooth.le		支持低功耗 (LE, Low-Energy) 蓝牙
android.hardware.camera2		新的摄像头 API
android.media.[browse/projection/session/tv]		支持多媒体浏览和电视
android.service.voice		支持语音解锁设备 (比如说一句 “OK Google” 设备就解锁了)
android.system		uname()、poll(2) 和 fstat[vfs](2)
android.service.carrier	22	支持 SMS/MMS (CarrierMessaging 服务)
android.hardware.fingerprint	23 (M)	支持指纹采集器
android.security.keystore		密码学意义上的密钥生成和存储
android.service.chooser		应用 “深度链接” (deep linking)

1 原文如此, 疑为 RFC3550 或 RFC3551 之误。——译者注





实际上,Android 使用的所有框架都是被打包在设备的/system/framework 目录下的数个 Java 格式的\*.jar 文件中的,而在 L 版中则是被预编译进 boot.jar 文件中的。尽管 AOSP 是开源的,但直接从 JAR 文件中找出相关的包也非常容易——只要调用 dexdump (或者 dextra 工具) 直接分析 JAR 文件中的 classes.dex 文件就行了。

## Dalvik 虚拟机

Android 对 Linux 另一个值得注意的扩展就是引入了 Dalvik 虚拟机。Dalvik 虚拟机是让 Android 能够在 256M 内存就已经算“很大”了的移动设备中正常工作的关键因素。Dalvik 并不是第一种试图能够运行在移动设备上的虚拟机——Sun 公司的 Microsystems 曾经被认为有望能够压过 Java 2 移动版 (J2ME) 一头,但实际上收效甚微。

Dalvik 主要是由 Dan Bornstein 发明的——他 2008 年在谷歌 I/O 大会上的演讲被认为是了解 Dalvik 虚拟机设计的一份很重要的参考资料。虚拟机的名字 (Dalvik) 是为了纪念冰岛北部的一个小渔村 (见图 1-3)。

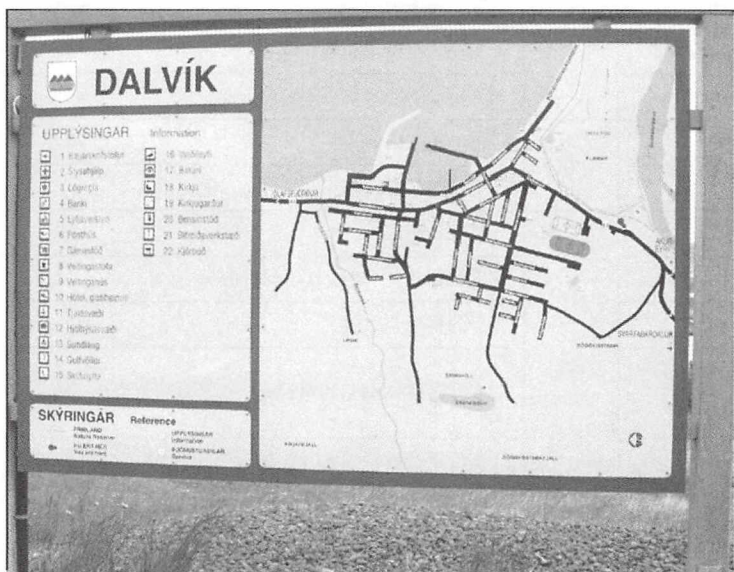


图 1-3 Dalvik 小村的地图, 由作者亲自拍摄

Dalvik 虚拟机尽管看上去和 Java 是等价的,但实际上并不是一个 Java 虚拟机。虽然偏离得并不是很远,但它运行的是一种完全不同形式的字节码 (这种字节码叫作 DEX, 也就是“Dalvik EXecutable”的缩写), 而且相对于 Sun/Oracle 设计的 JVM, 它在执行效率和共享内存方面做了更多的优化。这些优化使得它在受到严格限制的移动平台上占尽优势——这一点也正是 Java (特



别是 J2ME) 没法在有限的实现之外进一步获得增长的原因。

Android 选择 Apache 的 Harmony 文件的一个子集作为它的核心类 (core class) 的基础。之所以选择 Harmony 是因为它是免费的 (在 Apache 许可协议下) (原来是 Sun 的, 现在是 Oracle 的) JVM 的开源克隆体。Oracle 于 2010 年将谷歌告上了法庭, 理由就是谷歌从未正确地获得 Java 类库的使用授权, 这场旷日持久的官司甚至直到 2015 年初还没有了结。

截至本书付印时, Dalvik 虚拟机正在被 ART (Android 运行时, Android RunTime) 逐步取代, 正如本章稍后部分讨论的那样。与流行的观点恰恰相反的是, 这并不意味着 Dalvik 正在走向消亡。因为 Dalvik 只有在即时编译 (JIT, Just-In-Time compilation) 方面的部分会被取代, 而它使用的 (DEX) 文件格式作为至关重要的体系结构概念, 仍是非常有生命力的。因此, 我们将在第 2 本中详细讨论 Dalvik 和 ART。

## JNI

Android 应用是运行在虚拟机里的, 但有时 [通常是在需要访问硬件或其他设备 (或芯片集) 特有的功能时] 它还是需要执行虚拟机之外的代码的。所以 Dalvik 允许应用通过 Java Native Interface (JNI) 使用原生代码库 (ELF 共享库) 中的代码。

从某种程度上说, Android 对 JNI 是又爱又恨。厂商们无疑更青睐于 “纯” Dalvik 的应用, 因为所有的代码都是运行在虚拟机里的, 所以不会受到虚拟机/操作系统是运行在什么体系结构的处理器上的影响。在这种情况下, Android 应用可以在无须任何修改的情况下运行在任意一个平台上, 无论是 Intel、ARM、MIPS 还是其他什么处理器上。但是另一方面, 虚拟机环境也并非没有限制 (特别是在开发者非常关心图形处理问题时) 和缺陷 (特别是它很容易被反编译)。因此在应用中使用 JNI 以优化性能或对抗逆向工程的情况也是屡见不鲜的。有鉴于此, 谷歌也提供了使开发者能够生成原生库 (及二进制可执行程序) 的原生代码开发包 (NDK, Native Development Kit) (NDK 可以在 Android 开发者网站<sup>[14]</sup>上下载)。

并非所有的应用中都使用了 JNI, 但在那些使用了 JNI 的应用中, 我们也可以很方便地在安装包 (.apk 文件) 中找到 JNI 库, 因为它们是被放在一个单独的文件夹 “/lib/architecture” 中的<sup>1</sup>。DropBox 这个应用<sup>2</sup>就是一个较好的例子, 我们可以看到它至少为四种不同体系结构的处理器提供了支持代码 (下面给出的这个输出结果 1-1 是在一台 Galaxy Tab 3 10.1 上得到的)。

1 其中的 “architecture” 应该被替换为对应处理器体系结构的名称。——原注

2 别和同名的商用 App 搞混了, 这个应用是用来支持云存储的。——原注



```
root@Tab3:/ # % unzip -l /system/app/Dropbox.apk | grep lib/  
17532  05-14-13 23:11  lib/armeabi-v7a/libDbxFileObserver.so  
17528  05-14-13 23:11  lib/armeabi/libDbxFileObserver.so  
9352   05-14-13 23:11  lib/x86/libDbxFileObserver.so  
71076  05-14-13 23:11  lib/mips/libDbxFileObserver.so
```

输出结果 1-1 某个 APK 中使用的 JNI 库

通常情况下，尽管不同版本的 ARM 处理器确实也需要使用不同的库，但 JNI 代码在各种不同 ARM 设备（ARM 处理器也有不少版本）中的工作是不会有什么差异的（所以在上面这个输出结果中也有“armeabi”和“armeabi-v7a”这两个不同的文件夹）。不过当 Android 遇到 x86 体系结构的处理器时，JNI 就令 Intel 头痛欲裂了，而且可以预见的是会有越来越多的厂商想在使用 x86 处理器的设备上运行 Android。除了寄希望于应用的开发者会提供一份 for x86 的共享库（.so 文件）之外（大多数情况下开发者不会这样做），Intel 还提供了一个名为 Houdini 的闭源 ARM 模拟器（它对 Dalvik/ART 做了一些扩展，这一点我们将在第 2 本中予以讨论），可以作为他们的 Android 发布版本的一部分。这个模拟器（需要对 Dalvik 虚拟机做一些细微的调整）使得 ARM 原生代码也能运行在 Intel 体系结构的处理器上。

## 原生二进制可执行文件

从 Linux 的角度讲，所有的可执行文件都是 ELF 二进制可执行文件。Android 中的关键系统组件都是用 C/C++ 编写，并被编译成原生的二进制可执行文件的。而用户的应用则是编译成 Dalvik 字节码的，但字节码是运行在 Dalvik 虚拟机的上下文环境中的（或者在 ART 中，是在运行之前被编译成原生代码的），而 Dalvik 虚拟机本身也是一个 ELF 格式的二进制可执行文件。因此，尽管大多数开发者大可以心安理得地忘掉“二进制可执行文件”这么一回事，但这些二进制可执行文件仍在 Android 中扮演着重要的角色。

在 Android 中，二进制可执行文件通常都被放在 /system/bin 和 /system/xbin 这两个目录中（当然，还有一些重要的二进制可执行文件是放在 /sbin 目录中的）。由于它们本身就是 AOSP 的一部分，所以无论是在哪种设备中，大多数的二进制可执行文件都是一样的。但是厂商或者芯片集的制造商往设备里添加一些额外的二进制可执行文件的情况也不少见（比如，在高通的 MSM 多核设备中）。

你可以随时执行 ps 命令<sup>1</sup>，查看通过加载二进制可执行文件而运行起来的进程的列表。下面这个输出结果 1-2 中给出的就是这样一个例子（取自一台 HTC One M8 手机），我已经把其中

---

1 对该命令的执行结果要做一些过滤，以便把 App（应用）的进程过滤掉，因为这些进程是从 Zygote 进程中 fork() 出来，然后再加载了 Dalvik 类而产生的。——译者注





源自 AOSP 的二进制可执行文件高亮显示出来了。

```
shell@htc_m8wl:/ $ ps | grep " / " | cut -c1-22,55-
root      1      0      /init
root     218      1      /sbin/ueventd
root     365      1      /sbin/healthd
system   367      1      /system/bin/serviceanager
root     368      1      /system/bin/vold
radio    369      1      /system/bin/rild
system   370      1      /system/bin/surfaceflinger
root     371      1      /system/bin/pnpgcr
nobody   373      1      /system/bin/rmt_storage      # QCOM specific
radio    375      1      /system/bin/qmuxd            # QCOM specific
radio    376      1      /system/bin/netmgrd
root     379      1      /sbin/cpd
root     380      1      /system/bin/netd
root     384      1      /system/bin/debuggerd
drm       387      1      /system/bin/drmserver
media    388      1      /system/bin/mediaserver
install  389      1      /system/bin/installd
keystore 390      1      /system/bin/keystore
shell    391      1      /system/bin/dumpstate
root     393      1      /system/bin/thermal-engine   # QCOM specific
root     395      1      /system/bin/memlock          # HTC specific
root     397      1      /system/bin/clockd           # HTC Specific
system   400      1      /system/bin/gseecomd         # QCOM TrustZone
root     404      1      /system/bin/cand
system   505      400    /system/bin/gseecomd         # QCOM TrustZone
media_rw 844      1      /system/bin/sdcard
system   847      1      /system/bin/time_daemon      # QCOM specific
root     848      1      /system/bin/dmagent          # HTC specific; QCOM DIAG
nobody   850      1      /system/bin/hvdep            # QCOM Quick charge support
system   851      1      /system/bin/wcnss_service    # QCOM WLAN
root     852      1      /system/bin/htc_ebdlod       # HTC Specific
root     868      852    /system/bin/logcat2
media    919      1      /system/bin/adspapcd         # QCOM Application DSP
root     1170     1      /system/bin/logwrapper
wifi     1171     1170    /system/bin/wpa_supplicant   G
media_rw 1631      1      /system/bin/sdcard
root     1637      1      /system/bin/mpdecision        # QCOM SMP Policy
shell    12277    12149  /system/bin/sh
camera   23853     1      /system/bin/mm-qcamera-daemon # QCOM camera support
```

输出结果 1-2 HTC One M8 中的原生二进制可执行文件

因为 ELF 是个标准的文件格式，所以你可以使用任何一种 Linux ELF 文件解析工具（比如 readelf、objdump 或者其他 binutils 工具集中的工具）分析 Android 的二进制可执行文件。Android NDK 在“toolchains/”目录中也提供了完整的工具集（使用交叉编译技术编译的，这样这些工具就能运行在移动设备上了），支持 x86、MIPS、ARM 以及（从 NDK r10d 起开始支持的）ARM64，如输出结果 1-3 所示。

```
# replace "arm-linux-androideabi-4.9" with "aarch64-linux-android-4.9" for 64-bit ARM
morpheus@Forge (~)$ ls $NDK_ROOT/toolchains/arm-linux-androideabi-4.9/prebuilt/linux-x86_64/bin
arm-linux-androideabi-addr2line  arm-linux-androideabi-gcc-4.6  arm-linux-androideabi-objcopy
arm-linux-androideabi-ar         arm-linux-androideabi-gcov     arm-linux-androideabi-objdump
arm-linux-androideabi-as         arm-linux-androideabi-gdb     arm-linux-androideabi-ranlib
arm-linux-androideabi-c++        arm-linux-androideabi-gprof   arm-linux-androideabi-readelf
arm-linux-androideabi-c++filt    arm-linux-androideabi-ld       arm-linux-androideabi-size
arm-linux-androideabi-cpp        arm-linux-androideabi-ld.bfd   arm-linux-androideabi-strings
arm-linux-androideabi-elfedit    arm-linux-androideabi-ld.gold  arm-linux-androideabi-strip
arm-linux-androideabi-g++        arm-linux-androideabi-ld.mcl   arm-linux-androideabi-true
arm-linux-androideabi-gcc        arm-linux-androideabi-nm
```

输出结果 1-3 寻找 Android NDK 中的 binutils 工具集





## Bionic

与 Linux 发行版中使用 GNU 的 LibC (GLibC) 作为它们的核心运行时 (也就是著名的 `libc.so`) 不同, Android 选用了它自己的 C 运行时库——Bionic。尽管谷歌宣称选择 Bionic 的理由主要是因为它的简洁性<sup>[15]</sup>, 但实际上合法性的考虑也占了很重要的位置。如果在 Android 中使用了使用 GPL (GNU public license, GNU 公共授权协议) 授权的 GLibC, 那么根据 GPL, Android 也就必须开源 (这有点像 Linux 内核中使用 GPL 的情形), 而这又是谷歌所要极力规避的<sup>1</sup>。而 Bionic 尽管也是开源的, 但它混合使用了 BSD 授权协议 (BSD 授权协议对使用相关软件的第三方软件的限制更少些) 和谷歌自己的授权协议。

### 被 Bionic 去掉的特性

抛开合法性问题不提, 相对于 GLibC, Bionic 也可以算是非常轻量级的了, 而且对于 Android 所要达成的目标, Bionic 也更为有效。Bionic 中去掉的下列这些特性的原因或是认为没有必要, 或是认为太过复杂。

- **精简对系统调用的支持:** 由于系统调用是使用得非常频繁的, 所以 Boinic 想要通过尽量减少对它们的封装的方式, 降低因使用系统调用而引起的开销。系统调用的 stub 函数是在 `bionic/libc/SYSCALLS.TXT` 的帮助下自动生成的, 甚至有些系统调用都不会被导出。
- **不支持 System V IPC:** 在 Boinic 没有导出的系统调用中, 有一部分是用来处理 UNIX System V 进程间通信 (IPC, Inter-Process-Communication) (`sem[ctl|get|op]`) 和共享内存 (`shm[at|dt|get|ctl]`) 的。这是由于 Android 的设计使然——去掉这种形式的 IPC, 而改用 Android 自己的 IPC (我们将在第 2 本中讨论 Binder 和 ASHMem)。
- **有限的 Pthread 功能:** 一方面, Bionic 本身就支持 Pthread (即不需要一个单独的 `libpthread.so`)。但是另一方面, Bionic 对 Pthread 的支持也不是很全面, 不少特性已经不再支持了, 其中最值得注意的就是现在已经不能使用的 `pthread_cancel()` 这个用来杀线程的函数了。对 `mutex` 的支持也被阉割掉了, 取而代之的是更为高效的内核快速 `mutex` (`futex(2)` 系统调用), 此外较为高级的 IPC 对象 (比如 `rwlocks`) 也已经不复存在了。
- **有限支持 C++:** 尽管确实是支持 C++ 的 (而且事实上 Android 的大多数代码也是用 C++ 写的), 但是异常处理已经不支持了。与之类似, STL (Standard Template Library, 标准

---

1 谷歌极力避免与 GPL 沾边, 事实上因为授权协议引发的问题还不止是 Bionic 这一桩。在其他组件中同样有这个问题 (比如 `udev`)。这是因为 GPL 协议严格要求使用该协议授权的代码的软件也必须是使用类似 GPL 协议的开源软件。而规避 GPL 使得谷歌给自己留了个后手——可以在将来随时让 Android 不再开源 (实际上谷歌已经对 Honeycomb 版的部分代码这样做过一次了)。——原注



模板库)也已经不再包含在其中了——尽管也不是严格地不让使用 STL,不过你得要手工添加一些代码才行(部分代码可以在 external/stlport 项目中找到)。

- **不再支持本地化和/或宽字符:** Bionic 原生只需要 ASCII——尽管通过 libutils.so 还是可以使用 Unicode 的。

去掉这些东西还是很有道理的,特别是考虑到 Android 中大部分代码是跑在虚拟机里的,而使用虚拟机本身就意味着不需要 App 的作者使用这些功能。例如,虚拟机有自己的线程管理和 Unicode 支持(通过 ICU)。不过,去掉这些特性还是会给原生代码的开发者带来一些麻烦,特别是那些想要把一些库从 Linux 移植到 Android 上来的程序员,我们在下面的讨论中还会专门讨论这个问题。

## Bionic 新增的特性

Bionic 也在标准 LibC 中新增了一些特性,这些都是为 Android 而特别优化的,这些新增的特性包括:

- **系统属性。**系统属性是 Android 特有的特性,它支持系统或应用在一个简明的由“键/值”构成的存储空间中记录各种配置和操作参数。它类似于 Java 中“属性”的概念(而且,事实上它是可以通过 Java 的 System.properties 访问的)。Android 极度依赖于这一机制,系统属性是在一段共享内存中实现的,系统中的所有进程都能以只读权限访问它,但只有 init 进程才有对它进行设置(写)的权限。我们将在第 4 章中讨论系统属性的实现机制。
- **硬编码写死的 UID/GID。**与传统的 UNIX 系统中依赖于 passwd 和 group 文件进行权限管理的方式不同,Android 中选择使用硬编码写死的 id,并模拟 getpwnam(3)及其相关函数的方法进行权限管理。我们只要考虑一下 Android 的安全模型,就会觉得这一做法也不无道理:Android 中每个应用都会被分配到它自己的 UID 和 GID(从 10000 开始),而且这些 ID 会被映射为方便人类阅读的格式 app\_uXXX(或者从 JellyBean 版起,这一格式改为:uXX\_aYYY)。此外 Android 把数字较小的 UID/GID(1000-9999)保留了下来,供它自己的子系统使用。我们将会专门讨论安全问题的第 8 章中详细讨论这些(和目录的访问权限一起定义在 android\_filesystem\_config.h 中的)AID。
- **内置了 DNS 解析。**Bionic 中整合进了“名称-IP”的 DNS 解析代码(在传统 Linux 中,这部分代码是放在 libresolv.so 中的)。Bionic 中使用的代码更为安全(源端口和 Query ID 都是随机的,这样做能有效地防范“生日攻击”),并且引入了一个新奇的特性——每个进程都可以有自己的 DNS 解析服务器列表。这使得特定的应用可以通过重新设置

`net.dns.pid` 系统属性的方式，抓取 DNS 解析的结果或者重定向解析 DNS 请求的服务器。DNS 配置本身也是存储在系统属性中的(`net.dns#`)。猜也能猜到，`nsswitch.conf` 这个 Linux 中配置 DNS 解析的方式、顺序(NIS 还是 LDAP)的文件，在 Android 中已经不再支持了，尽管 `resolv.conf` 还是支持的(它被存放在 `/system/etc` 目录中)。

- 硬编码写死的服务和协议。因为已经把原来 `libresolv.so` 做的事都给包圆了，Android 中就去掉了传统 UNIX 系统中通常是放在 `/etc` 目录下的 `protocols` 和 `services` 这两个文件，并用 Android 内部的 `__res_get_static()` 函数模拟了 `getservent(3)` 系统调用。其他的一些 API，比如 `getprotoent(3)`，也不再支持了。

## 代码移植中的问题

由于(Bionic 中)去掉了一些特性，这就给代码移植(特别是把 Android 中的代码移植到 Linux 中去时)带来了一些问题。如果这些问题都能被妥善地解决，那么理论上就可以把 Dalvik 虚拟机移植到 Linux 或其他操作系统上去(而且确实有些开发者已经在这么干了，详见第 2 本)，而且还能让台式机也能运行 Android 应用。

对于要把代码移植到/移植出 Android 系统的努力来说，Bionic 是最主要的障碍。在(对 Bionic)进行适当的扩展，使之兼容 GNU LibC 的过程中，上文所述的这些被去掉的或新增的特性，确实也意味着一些更为有用的特性(特别是多线程)将不可移植。不过，对于某些源码包来说，移植到 Android 中去，要做的只不过是用 NDK 重新再编译一遍源码而已。通过这种方法，许多以 tar 包的形式给出的软件也可以被移植到 Android 中，只需对 `configure` 脚本和 `Makefile` 文件做出些许调整就可以了。

记住：无论是 Android 还是 Linux，它们导出的都是同一套系统调用。所以当你发现，静态链接的二进制可执行文件通常都是完全兼容(当然得是在 CPU 体系结构相同的前提下)的这一情况时，也请别大惊小怪。静态链接的二进制可执行文件会把执行时所需的函数(不论它是某个动态库提供的函数，还是内核提供的系统调用)都一股脑儿地编译到可执行文件中。一个挺有名的例子就是(我们将会在第 2 本中讨论的)Intel 的 Houdini——一个在 for x86/64 的 Android 系统上提供的程序。另一个更为常见的例子是 `BusyBox`，这是一个多合一的二进制可执行程序，用它你能在 Android 中使用各种 Linux shell 命令[一个静态编译的 for ARM 的 `BusyBox` 二进制可执行文件可以直接复制到其他(使用 ARM 处理器的)嵌入式 Linux 系统中运行，而且几乎完全兼容]只有在一些很小的方面(比如，在显示 Android 的 AID 时)并不总能完全正常工作。

值得注意的是，Bionic 本身也有一些开放的问题(特别是记录在 `bionic/ABI-bugs.txt` 中的那些)都是些看上去影响还不怎么明显，但实际上可能会造成严重后果的数据类型，比如(截至本书编写时的) `time_t` (一种用 32 位数表示时间的数据格式，到 2038 年，它所能表示的时间范



围就到头了，届时可能会爆发出一些比较严重的问题）和 `off_t`（一种用 32 位数表示文件内部偏移量的方法，在大于 4GB 的文件中使用它，可能会引发一些 bug）。此外，Bionic 本身就是为 32 位系统优化的，当苹果的系统迁移到 64 位上去时，也会迫使 Bionic（事实上是 Android 的所有组件）迁移到 64 位上去，这已经在 Lollipop 版中造成问题了，我们将在本章稍后部分讨论它。

## Android 的原生库

除了 Bionic 之外，Android 中还含有其他一些重要的库，这些库提供了对 Dalvik、框架和系统进程的支持。这些库被放置在源码树中各个不同的目录中，所以我们根据这些库（在源码树中）的存放目录对它们进行分类。

### 核心（core）库

这些库都位于 `system/core` 目录中，它们的主要作用是封装内核中一些 Android 特有的特性，或是在用户态中实现一些额外的功能。这些库包括：

- **Libcutils**——这个库提供了一些便于用户使用的支持函数，以提供对内核导出数据（比如 `/proc/cpuinfo`）的封装，支持 `socket` 以及 `ASHMem` 之类的 Android 特有的一些特性。
- **liblog**——这个库封装了 Android 的 `/dev/log` 机制，提供了一个快速、有效、基于 `ring-buffer` 的日志机制。
- **libion**——这个库封装了在冰激凌三明治版中引入的 ION Memory Allocator。
- **libnl\_2**——这个库提供了对 Linux NetLink `socket` 机制的封装。
- **libpixelflinger**——这个库主要用于 `SurfaceFlinger`（Android 图形栈的核心部件，详见第 2 本）——名称中的“Flinging”指的是将两个或两个以上的输入合并在一起的操作。以图形画面为例，经过这一操作后，画面上的每个像素点上都是合并前各个输入的画面上对应点上的色彩混合的合成效果（可能是 `alpha-blend` 混色的）。
- **libsuspend**——这个库中提供了电源管理中某些（特别是在与休眠和挂起操作系统相关的方面）封装好了的函数。

其他次要些的库包括：

- **libdiskconfig**——这个库中提供了对磁盘（闪存）的配置以及分区管理方面的封装好了的函数。
- **libcorkscrew**——这个库中的函数是供 `debuggerd` 分析栈使用情况以及应用（可以带符号链接的）“tombstone”（Android 应用崩溃或者被系统主动杀掉时留下的类似于“崩溃转储”的东西）的。
- **libmemtrack**——在硬件模块的帮助下（如果有的话），提供进程内存的 `trace` 服务。



- **libmincrypt**——提供在数字签名过程中所需的基本的 RSA 和 SHA-[1|256]实现代码。
- **libnetutils**——提供简化的网卡配置功能，并且支持 DHCP。
- **libsync**——这个库中提供了内核中一些 Android 特有的特性的封装。
- **libsutils**——这个库中提供的函数主要是供 Framework[Client|Listener|Command]、Netlink[Event|Listener]、Socket[Client|Listener]和 ServiceManager 等系统实用程序(utility)使用的。
- **libzipfile**——提供对 zlib 的封装，以处理 zip 文件。Android 中 zip 文件的使用是非常广泛的——甚至应用的安装包(.apk 文件)也是一种特殊格式的 zip 文件。

## 用以支持框架的库

位于 frameworks/目录中的库，为 Android 中的各种框架提供了原生代码级的支持服务。尽管不是“核心”(core)库的一部分，但这丝毫不影响它们的重要性。这些库又可以进一步细分。

- frameworks/base/core/jni 目录中存放着非常重要的 libandroid\_runtime.so 文件，这个库提供了对 Dalvik 虚拟机中的 JNI 的底层支持。在这个目录中还存放着拥有超过 85 个(Dalvik 虚拟机级的)框架类的各种 JNI 组件。
- frameworks/base/services/jni 目录中存放着重要性毫不亚于上一个库的 libandroid\_servers.so 文件，这个库提供了对各种 Android 服务的底层 JNI 支持。
- frameworks/base/native/android 目录中存放着 libandroid.so，这个库提供了 assets、存储管理器(storage manager)等功能的原生代码接口。
- base/libs 目录中存放的库中，包括名为 libandroidfw.so 和 libhwui.so 的库。前者提供了诸如 zip 文件解析、asset 管理等各种烦琐杂物的支持服务，后者(通过 OpenGL 和 SKIA)提供了硬件加速的 UI 渲染功能。
- 还有一些位于 av 目录中的库，用来提供处理多媒体、音频和视频的函数。这些库包括：
  - ◉ 与摄像头硬件抽象层(Camera HAL)相关的库——libcamera\_client.so 和 libcamera\_metadata.so (参见第 2 本)。
  - ◉ 支持 DRM 框架的库(libdrmframework.so)——这些库是供 Android 数字版权管理机制使用的。
  - ◉ 支持多媒体的库——这些库包括 libeffects.so、libmedia.so、libnbaio.so、libmediaplayerservice.so 和 libstagefright.so。

在 av/services 这个子目录中还存放着一些为这些服务提供进一步支持的库，它们是 libcameraservice.so、libaudiofingerprinter.so 和 libmedialog.so。

- native/libs 目录中存放的库包括：
  - ◎ **libbinder**——提供用来支持 Binder 的各种函数，详见第 2 本的深入讨论。
  - ◎ **libdiskusage**——这是一个很小的库，用来提供查询目录所占空间大小等功能的函数。
  - ◎ **libgui**——提供 surface 之类的各种 GUI 抽象，工作在 libui.so 之上。
  - ◎ **libinput**——这个库提供的函数基本上是仅供 Android 输入栈使用的，详见第 2 本的讨论。
  - ◎ **libui**——提供与窗体和图形缓存相关的原生 API，供 surfaceflinger 使用。

在 native/子目录中还含有一个 opengl/文件夹，其中存放着 EGL 和 OpenGL ES 的相关库，详见第 2 本的讨论。

## 源自其他项目的原生库

Android 还需要使用一些源自其他项目的库。尽管我们是用它们在 Android 源码树上的名字称呼它们的，但实际上，它们是属于其他一些开源项目的，只不过使用它们让 Android 操作系统平添了不少强大的功能罢了。

在 Android 源码树中使用了超过 150 个这类源自其他项目的原生库。所以我也并不（敢）打算在本书中将它们全部列出。不过，在表 1-4 中我还是试图列出其中一些提供了重要支持的库。

表 1-4 Android 源码树中使用的源自其他项目的库

目 录	作 用
bluetooth	Bluetooth 库（libbluetooth.so），提供了用户态下的蓝牙功能
icu4c	libicuuc 和 libicui18n，用以支持 Unicode 及国际化
mdnsresponder	苹果的多播（Multicast）DNS（Bonjour 协议）——包括一个守护进程（mdnsd）和一个库（libmdnsd.so）
libselinux libsepol	支持 SELinux（仅在 JellyBean 及更高版本中才有）
skia	SKIA 2D 图形库（详见第 2 本的讨论）
sqlite	支持 SQLite3 数据库，SQLite 数据库是许多 Android 数据库的核心部件
svox	libttspeco 和 libttscompat，提供了 SVOX Pico 文字转语音引擎

续表

目 录	作 用
tinyalsa	最低版本的 ALSA（Linux Advanced Sound Architecture）库
webkit	webkit 浏览器的核心库，供 WebView 控件使用
zlib	zlib——为 gzip 或类似的压缩文件提供压缩/解压功能的库

注意，一旦被部署到设备上去之后，这些来自其他项目的库基本上就不能和 AOSP 中的库区别开来了，因为所有的库最终都是一起放在/system/lib 目录中的。同样，如果你的设备中还有一些厂商提供的库，它们也有可能被放在/system/lib 目录中（尽管根据规定，这些库应该被放在/vendor/lib 目录中）。

## 硬件抽象层

Android 会运行在许多不同种类的设备上（平板电脑、手机、机顶盒、跑步机等），底层硬件在适用范围和支持方面都有极大的不同。为了解决这个问题，Android 定义了一个硬件抽象层（HAL，Hardware Abstraction Layer），其目标是通过定义一个转换层，使对各种不同类型硬件的操作趋于标准化。硬件厂商可以在内核态中随心所欲地使用它们自己生产的设备的驱动程序，但是必须提供一个接口库（shim），以 Android（特别是 Dalvik）预期的方式与操作系统对接。硬件抽象层上定义了从 Android 的角度看来，摄像头、GPS、传感器以及其他的硬件组件应该是个什么样子。这并不会妨碍厂商扩展或修改硬件的功能，只需要厂商把接口库放到/system/lib/hw 目录里去就行了，然后 HAL（硬件抽象层）（也就是 libhardware.so）会自动加载它们。输出结果 1-4 中给出的就是 Galaxy S5 手机中使用的 HAL 接口库。

```
root@S5:/ # ls -l /system/lib/hw
.. 9448 2014-03-09 18:21 audio.a2dp.default.so # a2dp BT audio profile
.. 5308 2014-03-09 18:21 audio.primary.default.so
.. 116348 2014-03-09 18:21 audio.primary.msm8974.so
.. 17708 2014-03-09 18:21 audio.r_submix.default.so
.. 9476 2014-03-09 18:21 audio.umb.default.so # Audio over USB
.. 13552 2014-03-09 18:21 audio_policy.msm8974.so # Audio Policy
.. 1306732 2014-03-09 18:21 bluetooth.default.so # BT, AOSP stock
.. 280728 2014-03-09 18:21 camera.msm8974.so # Camera
.. 5412 2014-03-09 18:21 consumerir.default.so # Infra-Red
.. 17640 2014-03-09 18:21 copybit.msm8974.so # Hardware accelerated copy
.. 26260 2014-03-09 18:21 flp.default.so # Fused Location Provider
.. 21756 2014-03-09 18:21 gps.default.so # GPS
.. 9736 2014-03-09 18:21 gralloc.default.so # Graphics memory allocator, AOSP stock
.. 14328 2014-03-09 18:21 gralloc.msm8974.so # Graphics memory allocator, Qualcomm
.. 107820 2014-06-06 13:32 hwcomposer.msm8974.so # Hardware accelerated surface composition
.. 5308 2014-03-09 18:21 keystore.default.so # Cryptographic storage
.. 5308 2014-03-09 18:21 local_time.default.so
.. 65412 2014-03-09 18:21 nfc_nci.MSM8974.so # Near-Field-Connectivity
.. 5316 2014-03-09 18:21 power.default.so # Power Mgmt, AOSP stock
.. 21924 2014-03-09 18:21 sensorhub.msm8974.so # Sensors
.. 54640 2014-06-06 13:32 sensors.msm8974.so
```

输出结果 1-4 Galaxy S5 手机中的硬件抽象层接口库

显然，硬件抽象层是 Android 系统非常重要的一个方面。这既是因为它的表现形式与 Linux



的有一些不同，又是因为它在完成支持种类繁多的 Android 设备这一任务时的出色表现。因此在第2本中会有一个专门的章节来讨论它。

## Linux 内核

Linux 内核得益于它开源和免费授权的本质，为 Android 提供了一个极为出色的基底<sup>1</sup>。尽管它现在已经和 Linus Torvalds 最初的版本差了不止十万八千里，但是 Linux 内核仍在以不可思议的速度演进着——每个月，甚至每一周都会增加新的特性。Android 所能拥有的功能在很大程度上会受到内核的影响。较为有名的例子当数 zRAM 和对 64 位的支持。后者有助于解释为什么表 1-1 中 Lollipop 版的 Android 系统会有两个最低版本——因为 Linux 内核是从 3.7 版开始才支持 ARM64（AArch64）的。谷歌在选择内核版本时也会受到一些限制——因为它只能从 <http://www.kernel.org> 网站中选择那些被标为长期支持（long term）的内核版本。

与其他 Linux 内核相比，Android 内核被编译得稍有差异。因为配置文件生成时同时使用了 Android 的基础配置和默认内核发布版本中的建议配置模板（如 [source.android.com](http://source.android.com) 网站中 kernel 一节所示）。<sup>[16]</sup>

前文中已经提到过，Android 在内核中引入了一些它特有的特性（Androidism）。其中的一些被放在内核的 core 中，用条件编译语句 `#ifdef` 加了一层保护，而剩下的则被放在了 `drivers/staging/android` 目录中。（在 3.10 或更高版本中）这些 Android 特有的特性包括：

- **匿名共享内存（ASHMem）**——这是一种允许进程间共享内存的机制。应用可以打开一个字符设备节点（`/dev/ashmem`）并创建一段内存空间，然后再把它映射（map）到进程内存中去。这需把工作限定在非全局可写（no world-writable）的目录中，并进行 System V 进程间通信。
- **Binder**——Binder 是 Android 中所有进程间通信的关键，它源自于 BeOS。Binder 表现为一个所有进程都能够打开的字符设备节点（`/dev/binder`）。Android 中的各种服务都注册在 Binder 这里，客户端可以在 `servicemanager` 的帮助下连上相应的服务。Binder 提供了一个有效的高级进程间通信机制，在第6章中我们会对 Binder 进行一番讨论，在第2本中还会更深一步地讨论它。
- **Logger**——提供基于内核的 ring buffer，用以高速记录日志。Android 的日志并不是记录

---

1 事实上，种类繁多的各种 Linux 分支都是在这个内核之上演化出来的，它们包括三星的 Tizen、Jolia 公司的 Sailfish、“火狐”操作系统以及运行在智能手机上的 Ubuntu 系统。所有这些系统都是 Android 系统潜在的竞争对手——尽管至少截至 2015 年初，它们的市场占有率还是那么微不足道。——原注



在文件中的，而是由一个字符设备节点（/dev/log）来承载的。Android Lollipop 版中专门为此新增了一个用户态守护进程——logd，我们将在第 5 章中讨论这个守护进程。

- **ION 内存管理器**——ION 内存管理器是在冰激凌三明治版中被引入的，其功能是向内核驱动和用户态下的类似模块（通过/dev/ion）提供高效的内存分配服务。ION 替换掉了旧的 Android 特有的组件 PMEM，ION 的设计目标是为各种不同的 SoC 架构中的内存管理提供一个统一的标准。
- **内存不足时的进程终止器**——这个组件改进自 Linux 原有的 OOM（Out-Of-Memory）进程终止器——它会在内存不足时，杀掉一些进程，以释放内存空间。不过 Android 中的这个组件是使用启发式的方法来寻找要被杀掉的进程的，而 Linux 原有的进程终止器是以一种更具确定性的方式控制杀进程的行为的，而且还允许定义内存压力等级。Android Lollipop 版中为此专门增加了一个用户态守护进程 lmkd，详见第 5 章的讨论。
- **RAM Console**——这是一种保存内核崩溃时输出的诊断用数据（线程转储和最近的日志）的机制。不过较新版本的 Android 系统已经弃用了这一特性，转而采用 Linux 内核中自有的一个功能了（详见第 2 章的相关讨论）。
- **Sync driver**——这是最新引入的 Android 特有的特性，引入它是为了快速同步基元（primitive），它主要是用在 Android 图形栈（特别是 surfaceflinger）中的。
- **定时输出和 GPIO**——使得用户态程序在用户态空间里就能访问 GPIO 寄存器，并在间隔一段时间之后自动设置 GPIO 寄存器的值。它的主要客户端是设备中的振动器（vibrator）功能——框架（通过硬件抽象层）可以把一个数值（单位为毫秒）写入/sys/class/timed\_output/vibrator/enable 中，这样就启动了振动器，并让振动器振动了这个值规定的时间之后停下来。
- **wakelocks**——最初这是一个单独的用来控制电源管理并阻止内核进入休眠状态的 Android 特有的特性。不过 wakelocks 现在已经逐步被并入到内核自己的 wakeup source 机制中去了（在第 2 本中将会详细讨论电源管理机制，在本书的官方网站上有一篇相关内容的预览章节）。

## 1.3 Android 的衍生产品

### 谷歌官方的衍生产品

谷歌已经明确表示，希望 Android 能在所有类型的设备中（不光是在手机和平板电脑上）占据一席之地。为了实现这一愿景，谷歌发布了三款 Android 衍生产品。

## Android Wear

在苹果可能大力研发“iWatch”的谣言的刺激下，不出世人所料，谷歌在推出 KitKat 之后，马上匆忙地宣布了“Android Wear”。Android Wear 是专为可穿戴设备（截至本书编写之时，所谓的“可穿戴设备”只有手表这一种类型的设备，尽管理论上它泛指任何一种可以穿/戴的设备）优化的 Android 系统。在核心代码（core）层面上，Android Wear 和用在手机和平板电脑上的 Android 是一模一样的，但鉴于手表上只能搭载一个很小的显示屏，它的“home activity”（主界面）已经被替换为一个简单得多的接口了（如图 1-4 所示）。其中包括（通过点击谷歌的图标）输入语音命令（这个是重点）、通知和提示卡。有些可穿戴设备还支持滚动循环屏幕（“round screen”）。



图 1-4 Android Wear 的运行界面

可以认为 Android Wear(它在 ro.build.fingerprint 系统属性中被标为“clockwork”)是 Android 的“精简版”——所有不必要的框架和服务都被移除了，只留下了内存管理以及 CPU 相关的模块（因为电池续航力是推广可穿戴设备的一个主要限制因素）。如果我们把用在手机上的 KitKat 和改进自 KitKat 的 Android Wear 系统进行一番比较的话，马上就能发现，所有与通话相关的服务（phone、iphonesubinfo、simponebook、isms），以及 print、appwidget、backup、usb、wallpaper、device\_policy 和 drmManager 在 Android Wear 中都被删除了。同样，预装的应用所占的空间也从超过 180MB（有大约 60 个包）精简到不到 12MB（只剩 16 个包）了。留下的应用都是与手表功能相关的（ClockworkSetup.apk、ClockworkSettings.apk 和 PrebuiltClockworkHome.apk），或是能在小屏幕上进行操作的（换而言之，KitKat 系统中 com.android.\* 里的默认包，在 Android Wear 里就不存在或不加载了）。Android Wear 的 SDK 及配套文档已经发布，你可以在 Android 开发者网站<sup>[17]</sup>上下载到它们。

当前 Android 可穿戴设备的设计功能定位是用作功能更为强大的设备（比如手机或平板电脑）的辅助设备。它们之间唯一的连接途径是蓝牙，而且 Android Wear 中的大多数框架都只不

过是一个 stub，只能连接并调用手机中功能完整的对应框架来完成操作任务。作为较早的时候就在该公司生产的“Galaxy Gear”手表上采用 Android Wear 操作系统的厂商，三星公司现在也已经放弃 Android Wear，而转用它自己研发的 Tizen 系统了。Android Wear 暴露出来的电池续航力低下和功能过于有限的问题是促成三星做出这一决定的关键因素。

## Android Auto

就在苹果宣布了“CarPlay”，把 iOS 7 植入汽车后不久，谷歌也很巧地宣布了“Android Auto”（如图 1-5 所示）。它的设计目标与“CarPlay”惊人地相似：提供一个使用便利的接口，使用户在汽车里也能使用移动设备，访问诸如导航、音乐播放器以及（当然还包括）手机等有用的应用。和 Android Wear 一样，Android Auto 的重点也是语言命令和通知——不过这次不再是屏幕太小的原因了，而是为了解放你的双手。

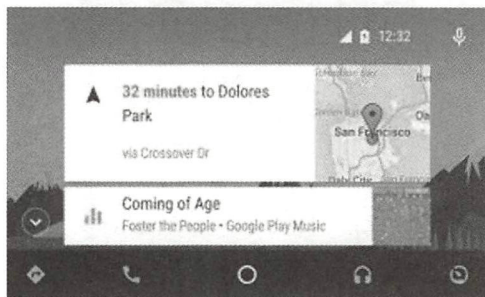


图 1-5 Android Auto 的界面（图片来源：谷歌）

从开发者的角度讲，最重要的区别在于：在 Android Auto 中，不需要专门开发一个汽车内专用的 UI。事实上，Android Auto 都可以不需要 UI——因为 Auto 中内置的系统 UI 是直接对接 App 中的对应功能的，并以“drawers”（一种在列表中列出所有候选项的菜单式显示方式）的形式显示出来。这样驾驶员就能直接用大多数方向盘上都有的上下键选中要使用的功能了。App 也可以对内置 UI 做一些定制，但这仅限于替换特定的图标或更改背景图片，而不需要（也不能）显示任何定制的 UI view——因为这些 view 都是通用的。开发者需要声明一个额外的 XML 文件，这个 XML 文件中应该含有一个 `automotiveApp` 元素（element），在这个元素中可以指定应用需要使用的特性，但当前只支持“media”和“notification”这两种特性。这个 XML 文件的需求记录在 `AndroidManifest.xml` 中的 `meta-data` 元素中，在其中的 `android:name` 中填上“com.google.android.gms.car.application”，在 `android:resource` 字段中填上开发者声明的 XML 文件。

谷歌在 Android Auto 网站<sup>[18]</sup>上详细给出了诸如 launcher、基于 drawers 的 UI 等接口的变化情况。



## Android TV

长久以来，电视机的制造商们在它们生产的电视机上运行的都是专用的操作系统，三星的 Tizen 和 LG 的 WebOS（也就是从前 HP 公司的 Palm）就是最有名的两个例子。谷歌也想要把 Android 的霸权扩展到这一领域（利用用户视觉习惯上的边际效应）。这是谷歌第二次企图进军电视机市场了——携其“Google TV”（一款略逊于“Apple TV”的产品）。

Android TV 是和 Android Lollipop 版一起被宣布并发布的，同时在 Android TV 网站<sup>[19]</sup>上还发布了丰富的文档。在模拟器中运行一下镜像，你就能感受到 Android TV 与普通 Android 系统的主要差异——从 launcher（com.android.mclauncher，位于/system/app/LeanbackLauncher.apk）到内置的 TV 应用（com.android.tv，位于/system/priv-app/TV.apk），再到 TV Content provider（com.android.providers.tv，位于/system/priv-app/TvProvider.apk）。Content provider 会导出 android.media.tv 的 URI 到 input（远程控制）、channel、program 和 watched\_program。后三者是存储在 provider 数据库（/data/user/0/com.android.providers.tv/databases）中的三张表。系统中其他的一些特性也经过了一定的修改，以适应 Android TV，特别是基于远程控制的导航系统和支持大屏幕。

Android TV 在将来很可能会得到可观的发展（特别是不断增加新的功能，紧跟苹果公司对 Apple TV 和 iOS 系统的更新）。未来可能新增的功能很有可能包括对音/视频流更好的支持、增强的 EPG（Electronic Programming Guide，电子节目菜单）功能、整合进 ChromeCast 以及支持游戏平台。不过在电视领域里还有另一个值得警惕的竞争对手——Amazon。

## 非谷歌官方的衍生品

由于 Android 开放的本质，厂商可以以各种方式定制 Android 系统——只有想不到，没有做不到。对标准 UI 所做的大多数增强（或者精简）的目的都是为了让消费者对厂商的产品不会产生“哦……谷歌又出了一款新的 Android 手机”的印象。比较有名的例子有 HTC 的“Sense”和三星的“TouchWiz” UI 界面。另一些厂商则是把 Android 用在新类型的设备上。比如英伟达把 Android 用在了它的“Shield console”电视盒子上。不过，在上述所有这些例子中，其基本系统仍然是和 Android 非常相似的。更进一步说，谷歌还（在 AOSP 的 cts/子树中）提供了“兼容性测试工具集”，为了得到谷歌官方的背书（并且确保各种 App 都能在修改过的系统上正常运行），厂商改装的 Android 系统都必须通过这一测试。此外，谷歌还把谷歌 Play 商店和其他服务紧紧地捆绑在一起。作为进入谷歌的 Android 生态系统的代价，厂商必须绑定谷歌的全套预装应用（谷歌地图、邮件等）。每个应用都使得用户在设备上绑定一个谷歌账号的可能性更大了一些——得！谷歌的标签可就又给打上了。



但仍有一些厂商仅仅是把 Android 作为一个基底，对系统做了大量的修改。它们更愿意放弃谷歌的 Android 生态系统——因为它们会自己构造一个新的生态系统出来。中国的智能手机生产厂商小米就是这样一个例子。它的旗舰版手机也只卖个白菜价——这让它成了中国（甚至可能是全世界）最大的智能手机生产厂商之一。小米通过投资建设它自己的生态系统，进而构建了一个完整的商业模式，因此在抛弃谷歌的服务时毫无心理压力。不难想象，谷歌对此一定是非常不爽的（这让它损失了至少一个亿的用户）。但这也只能算是 Android 的开源本质带来的必然后果。不过没有最倒霉，只有更倒霉。比如说诺基亚吧，在尝试了几个版本的 Android 之后，它已经“叛逃”到微软的云服务阵营里去了。而在大洋的彼岸，还有一个亚马逊（Amazon）呢……

## FireOS

亚马逊无疑是从 Android 中获利最多的厂商之一。巨型零售商的身份让它的 Kindle 产品线成功地涉足了平板电脑市场。Kindle 系列产品原本是基于一个拥有专利的嵌入式 Linux 发布版本的，搭载的是一块电子墨水（e-Ink）屏。从 Kindle Fire 平板电脑起，通过将 Android 用作它的核心操作系统，亚马逊成功地让它的平板电脑跟上了时代的潮流。

不过让谷歌更为气恼的是，亚马逊完全定制了它自己的 Android 版本，并把它改称为“FireOS”。UI 接口经受了完全的改装（用户用一种类似“旋转木马”的方式选择 App），Kindle 系列设备的 Boot Loader 是加锁的，只能接受来自亚马逊的指令。所以如果不用亚马逊账号，这种设备基本上是没什么用的。而谷歌的所有痕迹，包括谷歌搜索、谷歌 Play 商店、谷歌账号以及其他的所有东西，全都被连根拔掉了。

从技术的角度讲，FireOS 的核心部件仍然是很 Android 的。不过对它的修改依然是很激进的——包括删掉了所有谷歌的东西，并将其一一替换成亚马逊自己的应用，特别是：

- 旋转木马式的主界面——我们熟悉的 Android launcher 程序已经被替换成了亚马逊定制的 launcher，即 `com.amazon.kindle.otter`。
- 默认的浏览器改成了“Silk”——或者用它的另一个名字 `com.amazon.cloud9`。这是一个基于 WebKit 的浏览器，但经过了重度修改和优化，以便使用亚马逊的 EC2（Elastic Compute Cloud，弹性计算云）优化上网冲浪体验。
- 谷歌 Play 商店被替换成了亚马逊的 App 商店——这个商店在亚马逊系统内部中的称呼是 `com.amazon.windowshop` 和 `com.amazon.venezia`。
- 只能由亚马逊提供的屏保——亚马逊利用 Android 的“Dreams”功能，安装了一个全是广告的屏保。在系统内部，屏保功能是由 `com.amazon.dcp` 包中的数个组件实现的，而屏

保的内容则是存放在 `/data/securedStorageLocation/dtcp/` 目录里的（顺便提一句，亚马逊还修改了这个目录的访问权限——这能有效地防止用户禁用广告）。

- 强制性的 OTA 升级——`com.amazon.dcp` 包中含有一系列的服务，以确保设备能够经常不断地进行升级。与其他版本的 Android 系统不同的是：FireOS 从来不询问用户是否要升级——它只是不断地去检查是否有更新，如果有，就直接给你装上（自动升级这一话题将在第 3 章中展开讨论）。

亚马逊还从苹果的做法中借鉴了不少经验，其中最值得注意的是锁定（lock down）操作系统以防止 root（好吧，至少是试图阻止 root），以及在一旦安装了更新之后，阻止操作系统降级回旧的版本（而且安装更新这一操作通常还是自动完成的）。从 FireOS 的整体上来看，亚马逊在“去谷歌化”的道路上走得更远，甚至还发布了它自己的“Fire Phone”和“Fire TV”，而且这两个产品每个都有专门的接口和 API。

## 不带上层 UI 界面的 Android

拿到 Android 之后，删掉 Dalvik 及与之配套的框架，你就得到了一个既没有 GUI 支持，又不能用来构造/加入生态系统的操作系统。不过即使是这样一个操作系统，也仍然有其自身的价值。作为一个基本的嵌入式 Linux 操作系统，而且已经经过了修改，它已经能直接运行在 ARM 和 MIPS 处理器上了。在 Android 闪亮登场之前，嵌入式 Linux 是一摊超级复杂且差异性极大的烂摊子，这在很大程度上要“归功”于构建（build）系统时使用的交叉编译工具链及各种用户态库的复杂性。那些提供这种工具链和环境（并提供技术支持）的公司是非常抢手的。

但是 Android 完全打破了这一局面，砸了嵌入式 Linux 市场上主要商家的饭碗。相对于以前一个授权证书动辄几十万美元的天价，现在嵌入式 Linux 已经是完全免费的了。只要去下载 Android 源码和 NDK，任何人都能根据他自己的需求构建和定制出他自己的系统。这种以不带上层界面的形式部署的 Android 系统现在已经成了许多不需要 GUI 的系统（比如传感器、家用智能电器等等）的基础，而且很可能在“物联网”革命中扮演主要的角色。这种系统可以安装在使用 ARM 和 MIPS（甚至可能是 Intel）处理器的嵌入式设备上，被部署到任何可用的地方。

Android 可能可以在这两个世界里（一个是需要丰富框架的世界，另一个是不需要 UI 界面的世界）都混得风生水起。通过设置系统属性，即便没有 UI，系统也能完成特定的操作（比如与传感器对接），同时（开发者）也能享受到能够使用面向对象的编程语言以及 Dalvik 和 ART 环境的其他高级功能的好处。

## 1.4 对前方道路的思考

只有傻子才会预言将来，但是思考一下 Android 下一步的发展趋势还是蛮有意思的。如今 iOS 和 Android 之间的大战正如火如荼，Android 在不少特性上能够紧紧跟上 iOS（好吧，有些是公然抄袭，可能会引发官司），反之，iOS 也从 Android 身上借鉴了不少。但是从目前的现状出发，可以很明显地看出某些特性是很有可能会出现在下一版的 Android 里的（Macaroon？Meringue？鬼知道谷歌会用什么食品来命名它的下一代 Android 系统）。

### 兼容 64 位

iPhone 5S 发布时，苹果公司在这台手机上首次使用了 64 位移动处理器，这又一次使整个移动设备产业受到了震动，并使许多之前曾经轻率地认为 64 位处理器无助于市场销售的厂商感到困惑。在最一开始，64 位处理器不受重视的原因恰恰在于它的主要优势——大于 4GB 的内存地址空间的寻址能力。事实上，这一能力在移动环境中确实是有一定争议的。尽管目前有些平板电脑已经搭载了 2GB 的内存，但是 4GB 的内存仍然超出了大多数移动设备的实际需要。64 位的内存地址空间在访问效率上也略逊于 32 位地址空间（因为需要查更多的页表），所以很多厂商就开始嘲笑苹果已经“无力创新”，“只能搞个没什么用的骗人小玩意”了<sup>1</sup>。

不过事实却是，越来越多的搭载 64 位处理器的移动设备进入了人们的视野。尽管 ARM 64 位处理器上仍然能够跑 32 位的代码，但原生 64 位（ARMv8）的指令集已经为了提升执行效率而完全重写了。再加上更宽的 64 位寄存器（以及更大的寄存器集），这些优势很快就凸显出来。64 位体系结构的处理器（以及苹果在定制 A7 芯片时的一些卓越的设计）很快就让所有其他移动处理器的性能变得过时了，同时它还有令人难忘的低功耗的特点。事实上这也证明，之前炒作的那些“4 核”、“8 核”的概念才真是“没什么用的骗人小玩意”。更进一步说，在处理器中加入更多的核（core）也会直接影响电池续航力，所以大部分核在设备开机的大部分时间里实际上是处于休眠状态中的。今天，苹果旗舰产品中使用的处理器仍是 dual-core (A9) 或 triple-core (A9X)，却依然碾压移动工业界的其他同类产品。

事实证明，这一把处理器从 32 位升级到 64 位，而不只是在 32 位处理器的个别性能指标上升级的做法是个明智的举措，也是一个行之有效的策略。尽管 iOS 中所有的 App 都要重新编译后才能运行在 64 位的 iOS 系统中，而 Android 的 App 不用这样重新编译一遍就能直接跑在 64

---

1 对采用 64 位处理器提出批评最多的当数高通公司的高级副总裁兼 CMO，他宣称“他们（苹果）只是市场上耍了个小把戏，消费者不会从中得到任何好处”。但是一周之后，高通就撤回了他的评论，而他本人不久之后也被“调换工作”了<sup>[20]</sup>。——原注



位的 Android 系统中,但是把 Android 移植到 64 位仍然还是一个漫长的过程。这是因为 Android 的核心部件(特别是 Dalvik 和 Bionic)都是为 32 位而特别优化的,因此必须要完全重写才行。在所有的处理器生产厂商中,Intel 是最快搭上 64 位的顺风车的,因为它的移动处理器已经是完全原生 64 位的了。而各家 ARM 处理器的生产厂商对这一变化则需要一定的适应过程(尽管三星很快就宣布他们的“下一件大事”理所当然就是向 64 位进军)。HTC 的 Nexus 9 手机上搭载了第一块 64 位的 ARM 处理器(英伟达的 Tegra K1),高通公司也很快跟上脚步,发布了骁龙(Snapdragon)810 处理器(用在了 HTC One M9 上),三星也推出了他们自己的 Exynos(用在 S6 手机上)。QEmu(Android 模拟器中使用的是这家提供的处理器模拟模块)最终也进行了升级,在 Android M 版的预览版 SDK 中也支持了 ARM64 位模拟器。谷歌也宣布 Nexus 5X 和 6P(分别是 16 核和 8 核的)也是 64 位设备——这就坐实了 64 位作为新的工业标准的地位。

## ART (Android 运行时)

Android 在某些方面仍逊于 iOS——电源管理还不算是其中最差的那个。这个问题的根子还是在于 Android 的 Linux 血统上(Linux 系统是为不能被随便搬来搬去的台式机或者服务器设计的,在这一情景下,电源问题几乎不予考虑),但过细的分层也有不可推卸的责任。尽管分层提供了漂亮的抽象、可移植性以及优雅的设计等其他一些方面,但由于需要执行更多的操作,它们常常会造成性能和电源续航力的下降。Android 最重要的一层——Dalvik——的操作十分复杂,即使是经过了大量的优化(即 JIT 编译),仍然需要执行比直接执行原生代码多得多的操作。相对而言,iOS 的运行时和框架都是用 Objective-C(一种扩展了标准 C 的语言),使用的都是原生代码<sup>1</sup>。

Android RunTime (ART)就是一个能解决这一问题的替代方案。它在 KitKat 版中以“实验版”的名义被悄悄地引入 Android 系统中。ART 的设计目标是通过使用 AOT (Ahead Of Time, 预先编译技术)的编译,得到 LLVM 甚至是原生代码,从而让 Android 得到可以媲美 iOS 的性能。当前的 ART 相对于 Dalvik 只在电源及性能方面有一些微弱的优势(约 10%~20%),在某些测试中的表现甚至还不如 Dalvik。但是,从 Lollipop 版开始,ART 已经成为了一个可选的运行时了,并且在 Android 支持 64 位的过程中发挥了至关重要的作用<sup>2</sup>。

- 
- 1 在 iOS 8 中,苹果首次决定放弃 Objective-C,改用 Swift。这是一种轻量级编程语言,自夸在编译后或解释执行时拥有令人印象深刻的运行时性能提升。——原注
  - 2 Dalvik 的代码仍然是为 32 位,而不是 64 位优化的,这一点非常重要。尽管 Dalvik 确实支持“宽”(wide)数据类型,但它的大多数操作仍是 32 位的。这样就意味着,即使编译成原生代码后,虽然也能分享到一些 64 位处理器的优势,但代码的执行效率仍然是比不上“纯”64 位代码的。——原注



就像本章之前提到过的那样，无论如何，Dalvik 还远远没到需要彻底退出历史舞台的时候。（Android 中的）应用仍是以 Dalvik 字节码文件（classes.dex）的形式打包进 apk 文件的，而 ART 所谓的“编译成原生代码”，也只是要到等到 apk 文件被装到移动设备中去之后才开始进行的（也就只是替换了通常是由 dexopt 执行的设备端上的优化阶段罢了）。我们将在第 2 本在中详细讨论 Dalvik 和 ART。

## 多画面

---

让系统支持同时运行多个不同的 activity，并将它们的画面同时展示在屏幕上，在这一点上 Android 已经有了足够的动力：三星已经扩展了 GUI，以提供这一功能；Win 8 也支持这项功能；此外还有传说称，安装了 iOS 9.1 的 iPad 上也有这项功能。所以 Android 中出现这一主流技术也就不足为奇了。这只是一个纯框架级的特性，因为从 Linux 的角度看，并没有真正的实质性变化。运行中的 activity 实际上就是一个进程，各个不同的进程理所当然是能并发执行的。这同时也是把 Android 扩展为一个完整的台式机操作系统过程中的主要步骤。

## 把 Android 用作台式机操作系统

---

既然有这么多的平板电脑想要成为台式电脑的替代品，那为什么不把 Android 做成台式机的操作系统呢？微软发布了 Windows 8 操作系统，它既是台式机上的 Windows 操作系统，同时也改进了 Windows 对移动设备（平板电脑和手机）的支持。Android 也需要做出与之类似的反向转变，也就是让它的移动端操作系统能够支持台式机，让 Android 应用也能跑在台式机上。

这么做其实并不是很难，就像我们在第 2 本中会讨论的那样，Dalvik 的开源本性使它具有很高的可移植性。它在许多其他操作系统中[不光是在 Linux 上（显然啊），在 Windows、OS X 甚至是 iOS 上]都有对应的实现。当然这些 Dalvik 实现项目没有一个是谷歌发起或资助的。不过随着 iOS 和 OS X 走得越来越近，有人已经推测，在不久的将来，OS X 上将可以运行 iOS App（甚至还有人更进一步预测说，苹果公司将在它的 Mac 计算机上装上 ARM 处理器）。如果真的发生了这种情况，生态系统间的相互捆绑将会让一大波果粉彻底放弃 Android，而这显然是谷歌公司难以长期忍受的。

不过这事也不是一点障碍都没有。其中之一就是要支持所有的台式机上的应用程序。Linux 中的 OpenOffice 以及许多其他应用程序都已经构建在 X-Windows（和 GNOME 或 KDE）上了。所以要让它们也能在 Android 中运行还是要费上一番周折的。此外，还必须对 Android 做一些扩展，使之能够支持鼠标（虽然有争议说 Android 的 InputManager 已经支持光标设备了）以及多窗口（再说一遍，从技术上说，只要对 WindowManager 做一些扩展就可以支持这点了）。最后一个依

然重要的障碍是 ChromeOS，这个操作系统是谷歌开发的与 Windows 对阵的操作系统。谷歌希望它能够像 Chrome 打败 IE 成为世界上最流行的浏览器那样，取 Windows 之位而代之。

## Android 和 ARA 项目

ARA<sup>[21]</sup>是谷歌开发的一个项目的产品代码，这个项目的设计目标是生产完全模块化的智能手机（见图 1-6）。其基本思想是使所有的系统组件（CPU、显示屏、存储器）都可以相互交换，也就是都是可替换的。在 PC 机的世界里，类似的事实在平常稀松得很——给台式机装块硬盘，或者换个显卡实在是太正常不过了吧。ARA 是谷歌收购摩托罗拉移动事业部（不过后来又廉价卖给了联想）的残余项目。它是由前摩托罗拉 ATAP 部门（Advanced Technology and Projects division，高级技术和项目部门）研发的。这个部门在谷歌把摩托罗拉再次卖给联想时，被谷歌保留了下来。

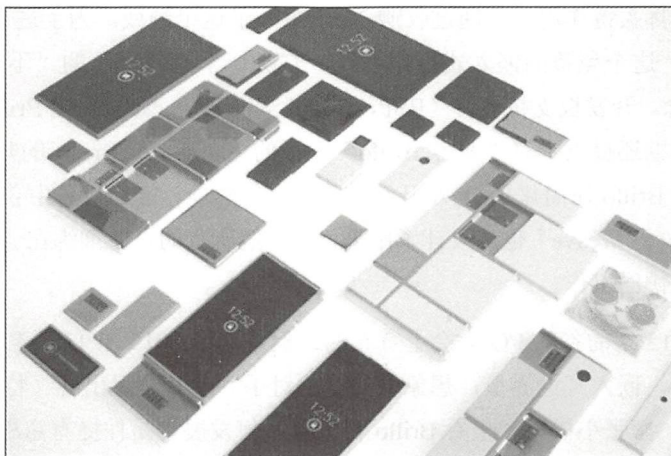


图 1-6 模块化的智能手机（图片来自摩托罗拉官方博客）

实际上，ARA 是把移动设备视作一个底架（更准确地说是一个骨架）加上一些组件（独立的模块）构成的装置。是不是和 PC 很像？用电永久磁铁（这种磁铁可以通过电子的方式控制开/关，但是正常使用时并不需要通电，这样才方便日常使用）把各个模块吸引在一起。理论上，所有的组件都是可以热插拔的（CPU 除外，可能显示屏也不行）。也就是说，在移动设备工作时，这些组件也可以被插上或拔下的。如果再加上 3D 打印技术，这甚至可以导致“可打印的”手机被设计出来。这种手机的设计图纸可以直接从网上下载下来，由一系列的可升级模块构成。如果真的出现，可能会导致我们现在用的这种为了能跟上时代的潮流而差不多每年都要换台新手机的设计模式全面消亡。

由于 ARA 是谷歌开发的，所以 Android 自然就成了它唯一可以选用的操作系统。不过，为

了支持 ARA，还是需要对 Android 进行一番大改造的。不光是在框架层，对底下的 Linux 层的改动甚至会更多，这样一路改下来，甚至连内核都需要做些修改。为了达成这一目标，谷歌已经和 Linaro 开展了合作，并且正可笑地在为所有标准化的模块开发必要的软件和硬件上投下大笔资金。截至本书第一次印刷时，ARA 仍然很不成熟，被大量的问题折磨着，它的发布也被推迟到了 2017 年（来自 2016 年谷歌 I/O 大会上的消息）。2016 年 9 月，路透社报道称，该项目已经被谷歌暂停了（但并没有说被砍掉），部分原因是“受到了谷歌精简硬件产品线的影响”。

不过，如果出现了某种奇迹，ARA 项目真的成功了，那么一台真正模块化的移动设备带来的影响，将不亚于第二次移动设备革命的来临。而这一次，谷歌希望它能够成为领头羊。

## Brillo

Brillo 是谷歌适应物联网（IoT, the Internet of Things）世界的项目产品代码。谷歌自认为将在这个世界中发挥关键作用，并通过收购 Nest 杀入了这个领域。为了进一步达成它的目标，谷歌提供了 Brillo 这个免费的嵌入式操作系统，它类似于之前讨论的“不带上层 UI 界面的 Android”这个概念。开发板支持套件（BSP, Board Support Package）让 Brillo 事实上能够胜任任何主板，而且谷歌还提供了对“兼容 Brillo”主板的免费支持。Brillo 的开发环境与 Android 的非常类似，而且 Brillo 还简化了 OTA 升级并且（很自然地）能从设备中获取大量的信息。一个专用的通信层——“Weave”也被设计了出来，用以把所有的设备捆绑在一起，并让它们能很方便地相互通信并协同工作。

Brillo 是在 2015 年的谷歌 I/O 大会上宣布的，但是截至 2015 年 11 月初，它仍然处在“仅邀请合作”的状态。嵌入式世界里，想象中的竞争对手并没有实际出现（特别是苹果对此没有任何回应）。不过，搬张小板凳，坐看 Brillo 将来会如何发展也是件挺有意思的事。

## 本章小结

本章探索了截至出版之日（KitKat 版）<sup>1</sup>Android 架构的历史变迁过程，重点讨论了它的底层特性。我们把 Android 的结构和它的亲缘操作系统 Linux 放在一起做了比较和对比，以示二者在许多方面都并非差得很远，尽管目前已经不能相互替换了。接下来，我们介绍了不少 Android 的衍生产品，尽管在表面和外观上有一些不同，但是它们在核心功能上都还是和 Android 一样的。因此你可以发现本书中所述的知识也一样适用于它们。本章最后还总结了 Android 进一步的发展方向（M 版及以后的版本），以及将来可能会支持（或者不支持）哪些特性。

---

1 原文如此，但 2016 年 11 月译者拿到手的新版的内容已经更新至 Nougat 版了。——译者注



在接下来的各个章节中，我们将尽可能详细地探索 Android 的方方面面。我们先从 Android 的文件系统着手，它自然是基于 Linux 的文件系统的，不过使用了一些预先定义好的分区和文件系统（一些相比其他更明确的定义）。

## 参考文献

---

- [1] Android version History, WikiPedia:  
[http://en.wikipedia.org/wiki/Android\\_version\\_history](http://en.wikipedia.org/wiki/Android_version_history)
- [2] Android Dashboards (usage statistics):  
<http://developer.android.com/about/dashboards/index.html>
- [3] Froyo feature summary: [developer.android.com/about/versions/android-2.2-highlights.html](http://developer.android.com/about/versions/android-2.2-highlights.html)
- [4] Gingerbread feature summary: [developer.android.com/about/versions/android-2.3-highlights.html](http://developer.android.com/about/versions/android-2.3-highlights.html)
- [5] Honeycomb feature summary: [developer.android.com/about/versions/android-3.0-highlights.html](http://developer.android.com/about/versions/android-3.0-highlights.html)
- [6] SMP Primer for Android: <http://developer.android.com/training/articles/smp.html>
- [7] Ice Cream Sandwich feature summary: [developer.android.com/about/versions/android-4.0-highlights.html](http://developer.android.com/about/versions/android-4.0-highlights.html)
- [8] Jellybean feature summary: [developer.android.com/about/versions/jelly-bean.html](http://developer.android.com/about/versions/jelly-bean.html)
- [9] Kitkat feature summary: [developer.android.com/about/versions/kitkat.html](http://developer.android.com/about/versions/kitkat.html)
- [10] Lollipop feature summary: [developer.android.com/about/versions/lollipop.html](http://developer.android.com/about/versions/lollipop.html)
- [11] Android version numbering convention: <https://source.android.com/source/build-numbers.html#platform-code-names-versions-api-levels-and-ndk-releases>
- [12] A much better Android architectural diagram:  
[http://source.android.com/images/android\\_framework\\_details.png](http://source.android.com/images/android_framework_details.png)
- [13] Dan Bornstein presenting Dalvik, Google I/O 2008: <https://www.youtube.com/watch?v=ptjedOZEXPM>
- [14] Android NDK: <http://developer.android.com/tools/sdk/ndk/index.html>
- [15] Google Groups Bionic discussion: <http://android-platform.googlegroups.com/attach/0f8eba5ecb95c6f4/OVERVIEW.TXT?view=1&part=4>
- [16] Android Kernel Configuration: <https://source.android.com/devices/tech/kernel.html>
- [17] Android Wear Developer Website: <http://developer.android.com/training/building-wearables.html>

[18] Android Auto Developer Website: <http://developer.android.com/training/auto/index.html>

[19] Android TV Developer Website: <http://developer.android.com/training/tv/index.html>

[20] Qualcomm reassigns exec after 64-bit criticism: <http://www.cnet.com/news/after-apple-64-bit-a7-criticism-qualcomm-exec-reassigned/>

[21] Project ARA official Website: <http://www.projectara.com>

# Android 的分区和文件系统

这一章一开始先讨论文件系统的基础——分区。设备中的存储器被划分成一些互不相交的部分，每个部分都会被单独格式化，用于不同的目的。我们将讨论该如何分析分区布局 (partition layout)，并研究许多保留的分区或不可访问的分区。

然后，我们转而讨论系统要正常使用的 Android 文件系统，如/system（安装操作系统的地方）、/data（存储用户数据的地方）等。我们会讨论这些文件系统的目录结构，并将重要的文件和目录一一列出。

最后，我们还会考虑 Linux 的伪文件系统——尽管它们本身不是 Android 系统的一部分（却是 Linux 内核的一部分），但是它们在系统操作的过程中（主要是在诊断和硬件访问时）还是提供了一些重要的功能特性。

## 2.1 分区架构

Android 的用户经常会诧异地发现：他们的设备往往宣称拥有“XXX GB 的闪存”，但实际可用的存储空间却远远小于广告上吹嘘的数量。高级用户连上 ADB，打入 df 命令，把列出的所有空间大小都加在一起（不光是 Android 操作系统占用的空间大小，还有被标为“used”和“free”的空间的大小），也依然达不到广告上承诺的存储容量。

其中的一些差异可以用用户协议中的一则条文解释。这一条文通常表述为：GB 的定义是模糊的，如果是按  $2^{10}$  进行转换的话，1KB = 1024 个字节，1MB = 1024KB，1GB = 1024MB，所以，1GB 就应该被定义为 1,073,741,824 个字节，而市场上只是把 1GB 定义为 1,000,000,000 个字节——这样广告上宣传的存储空间数据就一下子蒸发掉了 7%，这可不算是少了。可是扣除这一部分差额之后，二者之间的差距还是有不少，有时，甚至会有几百 MB 或更多的存储空间下



落不明。这些“遗失”的空间实际上是在设备的分区中被“吃掉了”。闪存中大多数存储空间都会被用于 Android，但是还有其他的一部分空间会被留作他用。Android 的闪存通常会被划分到许多个分区中去，而 Android 只使用了其中的大约 5 个分区。在某些设备中（比如 Kindle Fire 或者 Nexus 5），我们能看到 25 个甚至更多的分区，不过这也不能算是太稀奇，在 HTC One M8 中甚至有 48 个分区！但在所有这些分区中，用户只能往一个分区/data 中写数据，除此再无例外。事实上，在正常使用的过程中，这些分区中的大部分甚至都不会被 mount。在这一节中将讨论在存储器上划分分区的方法，以及使用工具找出其他隐藏的、常会含有我们关心的内容的分区的方法。



在不同的生产商甚至是不同型号的设备之间，Android 系统所使用的分区划分方式可能会有很大的差别。不过在大多数情况下，各个分区使用的语义是相同的，只是它们的大小和出现顺序会发生变化。本章中大部分例子是取自于使用高通芯片组（msm）的设备的，因为大多数 Android 设备都可以归为这类设备。

## 需要许多单独分区的原因

大多数桌面用户，特别是在 Windows 世界中，可能习惯于只拥有一个或者两个分区。经典的桌面视角需要的分区数总是比较少的，这一视角无疑更适合正规的 MBR 分区表示法。由于设计结果使然，这一表示法至多表示 4 个（主）分区。而在 UNIX 中，则把使用多个分区作为一种常态，因为在系统升级和进行其他管理员级的操作时，这一方案更为灵活。多分区确实也有一个明显的缺点，那就是由于可用空间要被分割到各个分区中，它对可用空间也就强加了人为的限制。不过 UNIX 管理员想出了解决这个问题的聪明法子——使用符号链接。即，在需要更多空间时，添加新的磁盘空间，并通过 mount 重定向它。

在移动领域里，使用多分区方案也很有必要——尽管是出于另外一些理由。移动设备中要考虑的主要问题之一是，它们必须总是可修理的（repairable），所以它们必须支持某种类型的“recovery”模式。为了能进行系统还原（recovery）或升级，必须能以某种方法，让系统从一个已知是“安全”的操作系统拷贝中启动。事实上，在一些设备上，有多个启动加载器（boot loader）组件（它们的内容完全一样，以便在万一的情况下，也能保证可启动性）的情况也并非罕见。另外，有些组件，比如 modem（调制解调器）或其他固件组件（以及启动加载器本身），也需要拥有自己的存储空间，以存储配置文件和镜像。

注意，并非所有的分区都会被 Android mount。事实上，只有很少几个分区会经常被 mount。剩下的要么是在 recovery 时才会被使用，要么只能被某些系统组件所使用。后者甚至会被设计成不能 mount 的形式，它们使用的通常是 Linux 内核不能识别的私有格式。

## GUID 分区表

把上述思考综合在一起，显然使用多分区就是必须的了。因为我们切实地认识到，其中的一部分需求是必须满足的。所以 MBR 分区表示法就直接出局了，剩下的 GUID 分区表示法(GPT)就成了可行的选项。从“技术上”讲，MBR 分区表示法也没完全被抛弃——设备的第一个扇区上一般含有一个“伪”MBR 记录，它把整个存储设备划分为一个分区。而在第二个扇区上记录的是 GPT 头，其中依次定义了所有的分区。如下面这个实验所示。

### 实验：从设备中获取分区表

由于其记录的位置处于所有分区之外，所以在用户模式下一般是访问不到 GTP 表的，因此我们需要从二进制底层去访问存储设备。如果你的设备已经 root 了，那么你只要把开头的几个扇区复制出来，就能（在你的主机上）使用 `file(1)` 命令去分析它并检查 GTP 表的内容了（见输出结果 2-1）。

```
# On device: use chmod as root to allow adb to read drive sectors
root@htc_m8wl:/ # chmod 604 /dev/block/mmcblk0
```

输出结果 2-1 读取和辨识 GPT



因为 adb 通常是以 `uid:shell` 权限运行的，所以直接读取扇区中的二进制内容有以下几种方式：

1. 以 root 权限重新运行 adb——这需要在启动过程中设置 `ro.secure` 和 `ro.debuggable` 属性，或者使用一个修改过的、不会降低权限的 `adb`。
2. 以 root 权限运行 `dd` 命令——为了把块设备节点中的数据复制到一个文件（比如你可以写入到 `/data/local/tmp` 文件）中去，并用 `chmod` 把它（块设备文件）设为 `uid shell` 可读。
3. 直接使用 `chmod`——在块设备节点上直接使用 `chmod` 命令，这样它就能被 `uid shell`（事实上是任意用户）读取了。请注意，在基于 SELinux 的系统，比如 KitKat 或更高版本的 Android 系统中，这一招可能会失效。

上述这些方法无一例外都会带来一定的安全风险，以 root 权限运行 `adb`，会使你的手机在落入坏人之后，变得毫无抵抗能力。不正确地使用 `dd` 命令（比如把 `if=` 和 `of=` 搞混了）会在瞬息之间把整个存储设备中的所有分区全给删掉。用 `chmod` 命令使设备对任何用户都变得可读的时候，至少会使得设备上潜伏的恶意应用能够访问到正常情况下应该受到良好保护的数据。

尽管第二种方法也常常被用来处理设备中二进制底层数据，但是最后一种方法才是本书中使用的方法。因为作者认为在这三种方法中，它引入的安全风险最小。首先，它是完全可逆的（而且还不需要重启）。此外，它只提供读权限（因此也就能避免误写操作而引发的问题）（见输出结果 2-2）。

```
# On host: Grab the first sectors of the drive, and identify with file(1)
morpheus@zephyr (~) % adb pull /dev/block/mmcblk0
^C # Hit CTRL-C before we copy all the drive image!
morpheus@zephyr (~) % file_mmcblk0
mmcblk0: x86 boot sector; partition 1: ID=0xee, starthead 0, startsector 1,
4294967295 sectors, extended partition table (last), code offset 0x0
morpheus@zephyr (~) % od -A x -t x1 mmcblk0 | more
# Sector 0 contains "protective MBR"
00000000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
00001c00  01 00 ee ff ff ff 01 00 00 00 ff ff ff ff 00 00
00001d00  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
00001f00  00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa
# Sector 1 (at 0x200 = 512 bytes) contains GPT
00002000  45 46 49 20 50 41 52 54 00 00 01 00 5c 00 00 00
#
E F I P A R T
00002100  71 00 18 78 00 00 00 00 01 00 00 00 00 00 00 00
00002200  ff df a3 03 00 00 00 00 22 00 00 00 00 00 00 00
00002300  ff ff 9f 03 00 00 00 00 32 1b 10 98 e2 bb f2 4b
00002400  a0 6e 2b b3 3d 00 0c 20 02 00 00 00 00 00 00 00
00002500  30 00 00 00 80 00 00 00 b2 e5 23 34 00 00 00 00
#
48 partitions
```

输出结果 2-2 用 file(1)识别存储设备扇区中的数据

完整地讨论 GPT 远远超出了本书的范畴，而且也没有必要。Linux 内核能够认出它表示的各个分区，并通过/proc（通常是/proc/partitions）把它导出到用户空间。我们将会在后文中讨论这一点。

## 闪存（Flash Storage）系统

Android 设备使用的存储系统并没有完全标准化，有些设备使用的是 MTD（Memory Technology Devices），另一些设备（比如 HTC One）使用的则是 eMMC（Embedded MultiMedia Card），还有一些设备中甚至还用着 MMC（MultiMedia Card）。根据系统具体使用的是什么存储设备，用户态中这些分区就会被映射（map）到各自对应的/proc/mtd 或/proc/emmc 以及 Linux 中标准的/proc/partitions（通常与前二者中的某一个一起使用）中。

在大多数情况下，具体使用的是什么系统对于大部分人来说几乎都是透明的。在高层，不同的存储系统之间的主要区别在于：MTD 是一个在 raw flash 之上的一个抽象层，而 MMC 和 eMMC 则有它们自己的 FTL（Flash Translation Layer，闪存转换层），在内核中表现为一个块设备。所以大多数较新款的设备中使用的都是 MMC 或者 eMMC，因为这类存储系统能和新款设备中所使用的基于块的文件系统（比如，继承自 YAFFS，作为新 Android 版本的可选文件系统之一的 ext4）配合得更好。

## 文件系统

Android 并没对该使用什么文件系统有过什么硬性规定，但由于 eMMC 和 MMC 在存储层



上是导出为一个块设备的，所以在使用这类闪存的设备上使用的都是 Linux ext4 文件系统（从“姜饼人”系统开始，替换了旧的 YAFFS 文件系统）。ext4 从 2.6.27 版开始已经成为 Linux 默认的文件系统，它是一个经过了良好测试的文件系统，尽管不一定是一个专为闪存优化的文件系统。

有些设备（特别是 Moto X）已经开始使用 F2FS（Flash Friendly File System）文件系统作为 data 分区的可选文件系统之一了。此外，Google 的 Nexus 中默认选用的文件系统也是这个。这个（由三星设计的）文件系统是个日志式结构（log-structured）的文件系统，专为 NAND 闪存优化，自称由它带来的性能提升，特别是在随机写请求上的性能提升超过了 Ext4。当然，由 XDA-Developers<sup>[1]</sup>进行的大范围内的基准测试表明，F2FS 确实对 Ext4 有明显的优势。

三星的论文<sup>[2]</sup>及 Neil Brown 发表在 LWN.net 网站上的一篇文章<sup>[3]</sup>对 F2FS 的特性进行了详尽的讨论。从 3.8 版起，它也已经被整合进了 Linux 内核主线，而且因为 Android 已经更新到了更高版本的内核，所以该文件系统有望在更多的设备上被使用。

Android 也支持 vFAT 文件系统，这是一种兼容 MS-DOS 的文件系统，常被用在 SD 卡上。因为它面向的是 DOS 和 Windows 9x 世界，所以 vFAT 是不支持权限这个概念的。因此，为了解决这个问题，Android 需要借助于辅助进程 mount SD 卡，通过一种特殊机制解决这一问题。详见本章后面的讨论。

内核在 `/proc/filesystems` 中给出了一张能被系统识别的所有文件系统的列表。这个伪文件中列出了所有系统原生支持的（即，被编译进内核的）或由加载的模块支持的文件系统。在 Android 内核中，生产商通常会把需要使用的文件系统的支持代码直接编译到内核中（尽管也可以通过把某个文件系统的支持代码放在一个模块中，然后把该模块绑定为 root 文件系统的一部分的做法来实现这一点）。

关于文件系统，一个好消息是，只要它们能够正常工作，用户就可以完全不用管到底使用的是什么文件系统。可坏消息是，万一文件系统不能工作了（特别是出现了文件系统关键数据被破坏了的情况时），文件系统的关键数据遭到破坏是非常罕见的情况（谢天谢地！），而且这些情况通常是在设备非正常关机（比如没电了或者非预期的崩溃）或在硬件出错的情况下才会发生。对于各种不同的文件系统，Android 分别提供了各自对应的默认检查和修复文件系统的二进制程序——`e2fsck`、`fsck_msdos` 和 `fsck.f2fs`。在文件系统（由 `init` 或 `vold`）被 `mount` 的时候，这些二进制程序会被自动执行。

## 实验：检查 Android 设备中的各个分区

你可以通过查看 `/proc/partitions` 文件中的内容，获知你 Android 设备的分区映射情况。这是一个标准内核 `/proc` 中的伪文件，其中记录了一张（系统中）所有块设备的列表。MMC 和 eMMC

设备的闪存层（flash storage layer）在这里以“mmcblk#[p#]”的形式显示，其中设备编号从 0 开始，分区编号从 1 开始。块（block）的大小为每个块 512 个字节（1/2 KB）。输出结果中的“major”和“minor”两列是指设备驱动。其中，“major”一栏实际上是驱动在内核的块设备表（block device table）中的索引号，而“minor”栏则是逻辑设备的索引号（在这个例子中，可以用来区分各个分区）（见输出结果 2-3）。

```
shell@nexus5$ cat /proc/partitions
major    minor    #blocks  name
179      0      15388672 mmcblk0    # Entire Flash disk size
179      1       65536 mmcblk0p1  # 1st Partition
...
179     29          5 mmcblk0p29 # 29th Partition
179     32      4096 mmcblk0rpmb # Resource Power Management backup
```

输出结果 2-3 Nexus 5 上的 /proc/partitions 中的内容

仅仅根据这些晦涩的名称区分各个分区还是比较困难的，不过万幸的是，大多数设备都有符号名称（by-name 或 by-num），它们位于 /dev/block/platform/name.# 目录中。其中的 name 就是控制器（controller）的名称，在使用高通处理器的情况下，msm\_sdcc.l 指的就是主存储器（见输出结果 2-4）。

```
shell@nexus5$ ls -l /dev/block/platform/msm_sdcc.l/by-name | cut -c56-
DDR -> /dev/block/mmcblk0p24
aboot -> /dev/block/mmcblk0p6      # Application Boot loader (Android Boot)
abootb -> /dev/block/mmcblk0p11    # Backup of Application Boot Loader
boot -> /dev/block/mmcblk0p19      # Kernel + InitRAMFS used to boot system
cache -> /dev/block/mmcblk0p27     # Mounted as /cache (used for updates/recovery)
crypto -> /dev/block/mmcblk0p26
fsc -> /dev/block/mmcblk0p22
fsg -> /dev/block/mmcblk0p21
grow -> /dev/block/mmcblk0p29      # usually empty
imgdata -> /dev/block/mmcblk0p17   # Boot loader graphic images (imgdata format)
laf -> /dev/block/mmcblk0p18       # LG Advanced Flash Daemon
metadata -> /dev/block/mmcblk0p14  # usually empty
misc -> /dev/block/mmcblk0p15      # Reserved for system->Boot Loader communication
modem -> /dev/block/mmcblk0p1
modemst1 -> /dev/block/mmcblk0p12  # Modem (Radio) boot
modemst2 -> /dev/block/mmcblk0p13
pad -> /dev/block/mmcblk0p7        # usually empty
persist -> /dev/block/mmcblk0p16   # Persistent settings for system components
recovery -> /dev/block/mmcblk0p20  # Alternate kernel + InitRAMFS for recovery
rpm -> /dev/block/mmcblk0p3        # Resource/Power Management loader
rpmb -> /dev/block/mmcblk0p10      # Backup of Resource/Power Management loader
sbl1 -> /dev/block/mmcblk0p2       # Secondary Boot Loader
sbl1b -> /dev/block/mmcblk0p8      # Backup of Secondary Boot Loader
sdi -> /dev/block/mmcblk0p5
ssd -> /dev/block/mmcblk0p23
system -> /dev/block/mmcblk0p25    # mounted as /system
tz -> /dev/block/mmcblk0p4         # ARM TrustZone
tzb -> /dev/block/mmcblk0p9        # Backup of ARM TrustZone
userdata -> /dev/block/mmcblk0p28  # mounted as /data
```

输出结果 2-4 Nexus 5 上的 //dev/block/platform/.../by-name

注意，在你手上的 Android 设备里，分区的名称有可能而且很有可能与上面给出的完全不

同。尽管大多数的 MSM 设备一般都会遵从上述约定,但是基于 NVIDIA 的设备就不是这样(在这种设备上,是用非常难懂的三个字母缩写表示各个分区的名称的),基于 OMAP 的设备也一样。

## Android 设备中的分区

如上一节中实验的结果所示,Android 设备中的各个分区都是有名字的,但是这些名称大多十分晦涩。使情况进一步复杂化的另一个问题是,不同的芯片组和制造商还会使用不同的分区,甚至会给功能完全一样的分区取不同的名称。为了讲清楚这一问题,我们把 Android 中的所有分区划分成下面几种类型。

### 标准 Android 分区

在所有的设备中,这类分区都有一个共同的特性:它们是被硬编码到 Android 系统自身中的,会出现在源码树的各个不同的位置上。这些分区构成了操作系统的核心(core)。

标准分区基本上都是可以 mount 的,仅有 boot 和 recovery 分区是例外,这两个分区是用 Android 专用的 bootimg 文件系统(详见下一章)格式化的。表 2-1 中列出了这些分区。

表 2-1 Android 的标准分区

名 称	文件系统	注 释
boot	bootimg	内核 + initramfs。含有内核和默认启动过程中所需的 initramfs
cache	Ext4	Android 的/cache 分区: 用来进行系统升级或 recovery
recovery	bootimg	用于把系统启动到 recovery 模式下: 内核+把系统启动至 recovery 模式的 initramfs
system	Ext4	Android 的/system 分区——存放操作系统的二进制可执行文件和框架
userdata	Ext4/F2FS	Android 的/data 分区——存放用户数据和配置文件

Android 设备中有一张文件系统 mount 表,这张表位于/system/etc/vold.fstab 或者(在版本更高一些的 Android 系统中)/fstab.hardware 目录下,在系统启动过程中会被 vold(Volume Daemon, 卷守护进程)加载。该表指出了哪些分区应该被自动 mount 上来(其作用类似于经典的 UNIX 系统中的/etc/fstab)。

### 芯片组专用分区

芯片组制造商经常需要一些分区用来存储支持芯片组工作的程序和数据。这里最著名的例子当属高通了。它的 MSM 芯片组(可以说是移动设备用得最广泛的芯片组)使用的分区如表 2-2 所示,其中的 bootldr 文件系统将在下一章中讨论。



表 2-2 在使用高通 MSM 芯片组的设备中找到的分区

名 称	文件系统	注 释
aboot	bootldr	Application Processor Boot, 其中含有 Android 启动加载器 (Boot Loader)。注意, 有些设备会用定制的启动加载器 (比如 HTC 的 HBoot) 把它替换掉
modem	MSDOS	含有一些用以支持设备调制解调器正常工作的 ELF 格式二进制程序和数据文件
modemst[1 2]	专用文件系统	调制解调器的断电不丢失 (Non-Volatile) 数据
rpm	ELF 32 位	资源电源管理 (Resource Power Management), 它提供了第一阶段的启动加载器 (Boot Loader)
sbl[123]	专用文件系统	次级 Boot Loader (secondary boot loader), 这是移动设备启动时由 boot ROM 加载的代码, 它执行完毕之后会去加载 Boot Loader, 根据具体设备型号的不同, 这个 sbl 可能会被分为三个阶段——sbl1, sbl2, sbl3 <sup>1</sup>
tz	ELF 32 位	ARM 可信区域 (TrustZone)

## 厂商专用分区

能在 Android 设备上找到的其他剩下的分区都是厂商专用的。它们根据自己的目的 (大多用于设备配置维护和升级操作) 来使用这些分区。这些分区使用的文件系统格式多为专用格式, 表 2-3 列出了这类分区中的一部分。

表 2-3 厂商专用分区

名 称	厂 商	注 释
hboot	HTC	HTC 专用的启动加载器, 在 HTC 设备中替换了 aboot 分区
efs	三星	加密文件系统。含有多个配置文件
ssd	三星	用于软件安全下载 (Secure Software Download)
ota,fota	三星	Firmware-Over-The-Air, 在手机升级过程中会被使用
grow	三星,LG	空分区, 用于增加分区容量
laf	LG (G2,Nexus5)	含有一个设备刷机时使用的, 加载 lafd (LG 高级闪存守护进程, LG Advanced Flash Daemon) 的可选 booting。laf 分区使用的是 recovery 镜像格式
imgdata	LG (G-PAD,G2,Nexus5)	IMGDATA 格式的 RLE 镜像, 类似 BOOTLDR

XDA-Developers 论坛中维护着一个不断更新的不同设备中分区映射 (mount map) 情况的

<sup>1</sup> 关于 sbl 的讨论详见第 3 章。——译者注

列表（类似上一个实验中给出的结果），该列表名为“El Grande Partition Table Reference”<sup>[4]</sup>。

### 实验：观察设备中已经 mount 了的分区

你可以使用 `df` 或 `mount` 命令查看所有的 mount 点。前者会给出各个已 mount 了的分区的使用情况的统计信息。输出结果 2-5 显示了用 `df` 命令列出的 mount 信息，执行 `df` 命令并输出结果的是一台安装了 Android L 系统的 Nexus 9。

```
shell@flounder:/ $ df
Filesystem      Size      Used    Free   Blksize
/dev            918.0M    32.0K    917.9M   4096
/sys/fs/cgroup  918.0M    12.0K    918.0M   4096
/mnt/asec       918.0M     0.0K    918.0M   4096
/mnt/obb        918.0M     0.0K    918.0M   4096
/system         2.5G      1.6G    875.0M   4096
/vendor         245.9M    149.3M    96.6M   4096
/cache         248.0M    256.0K    247.7M   4096
/data           11.0G      1.5G     9.6G   4096
/mnt/shell/emulated 11.0G      1.5G     9.6G   4096
/storage/emulated 918.0M     0.0K    918.0M   4096
/storage/emulated/0 11.0G      1.5G     9.6G   4096
/storage/emulated/0/Android/obb 11.0G      1.5G     9.6G   4096
/storage/emulated/legacy 11.0G      1.5G     9.6G   4096
/storage/emulated/legacy/Android/obb 11.0G      1.5G     9.6G   4096
```

输出结果 2-5 在一台 Nexus 9 上运行 `df` 命令的例子

注意，由于这条 `df` 命令是在 `toolbox` 工具中实现的，所以它的输出结果与传统 Linux 中 `df` 命令的输出结果有所不同——它只显示位于真正的存储器（也就是“真实的”存储设备）上的 mount 了的文件系统。

与之相反，使用 `mount` 命令会给出更详细的信息，因为它会显示出（实现在内存中的）伪文件系统及所有其他 mount 点的相关信息。不过代价是输出结果很占屏幕空间。`mount` 时使用的参数也会被一并显示出来，这些参数有些是通用的，有些则是只能用在特定的文件系统上的，表 2-4 中解释了你可能会遇到的各个 mount 参数（mount option）的作用。

表 2-4 常见的 mount 参数

参 数	类 型	作 用
ro	通用	只读：只允许读操作，不能修改
rw		读写：允许读和写操作
acl	通用	为了能比 user/group/other 权限设置方式更好地进行控制，允许扩展访问控制列表
seclabel	通用	在文件系统中启用 SELinux 标签
nosuid	通用	文件系统会忽略二进制可执行文件的 SetUID 权限位

续表

参 数	类 型	作 用
noatime	通用	不会记录文件访问操作的时间，只记录文件的创建和修改时间。这使得文件访问速度更快，降低了向文件系统写入数据的频率
relatime	通用	文件的创建和修改时间也会被视为文件的访问时间被记录下来
data=	专用于 Ext3/4	Ordered: 数据在它的元数据被提交给日志之前，就会被直接写入文件系统 Journal: 在写入文件系统之前，所有数据都要先被提交到日志中
errors=	专用于 Ext3/4	continue: 静默地忽略错误 remount-ro: 在出现错误时，把文件系统重新 mount 成只读模式 panic: 令系统崩溃
Background_gc	专用于 f2fs	在一个内核线程中，回收被删除文件释放出来的空间

mount 命令的执行结果如输出结果 2-6 所示。伪文件系统用斜体标出<sup>1</sup>，而且由于它们不是通过某个设备 mount 上来的，你也可以很方便地认出它们来，即伪文件系统所在行的第一列不是一个以/dev 打头的路径。

```

shell@flounder:/$ mount
rootfs / rootfs ro,seclabel,relatime 0 0
tmpfs /dev tmpfs rw,seclabel,nosuid,relatime,mode=755 0 0
devpts /dev/pts devpts rw,seclabel,relatime,mode=600 0 0
none /dev/cpuctl cgroup rw,relatime,cpu 0 0
adb /dev/usb-lfs/adb functions rw,relatime 0 0
proc /proc proc rw,relatime 0 0
sysfs /sys sysfs rw,seclabel,relatime 0 0
selinuxfs /sys/fs/selinux selinuxfs rw,relatime 0 0
none /sys/fs/cgroup tmpfs rw,seclabel,relatime,mode=750,gid=1000 0 0
pstore /sys/fs/pstore pstore rw,relatime 0 0
/sys/kernel/debug /sys/kernel/debug debugfs rw,relatime,mode=755 0 0
none /acct cgroup rw,relatime,cpuacct 0 0
tmpfs /mnt/asec tmpfs rw,seclabel,relatime,mode=755,gid=1000 0 0
tmpfs /mnt/obb tmpfs rw,seclabel,relatime,mode=755,gid=1000 0 0
/dev/block/platform/sdhci-tegra.3/by-name/APF /system ext4 ro,seclabel,relatime,data=ordered 0 0
/dev/block/platform/sdhci-tegra.3/by-name/VNR /vendor ext4 ro,seclabel,relatime,data=ordered 0 0
/dev/block/platform/sdhci-tegra.3/by-name/CAC /cache ext4 rw,seclabel,nosuid,nodev,noatime,
errors=panic,data=ordered 0 0
/dev/block/dm-0 /data f2fs rw,seclabel,nosuid,nodev,noatime,background_gc=on,user_xattr,acl,
errors=panic,active_logs=6 0 0
/dev/fuse /mnt/shell/emulated fuse rw,nosuid,nodev,relatime,user_id=1023,group_id=1023,
default_permissions,allow_other 0 0
tmpfs /storage/emulated tmpfs rw,seclabel,nosuid,nodev,relatime,mode=050,gid=1028 0 0
/dev/fuse /storage/emulated/0 fuse rw,nosuid,nodev,relatime,user_id=1023,group_id=1023,
default_permissions,allow_other 0 0
/dev/fuse /storage/emulated/0/Android/obb fuse rw,nosuid,nodev,relatime,user_id=1023,
group_id=1023,default_permissions,allow_other 0 0
/dev/fuse /storage/emulated/legacy fuse rw,nosuid,nodev,relatime,user_id=1023,group_id=1023,
default_permissions,allow_other 0 0
/dev/fuse /storage/emulated/legacy/Android/obb fuse rw,nosuid,nodev,relatime,user_id=1023,
group_id=1023,default_permissions,allow_other 0 0

```


输出结果 2-6 在一台 Nexus 9 上运行 mount 命令的结果

1 原文如此，可是在下面这个输出结果中没有用斜体标出来啊！——译者注



## 2.2 Android 文件系统中存储的内容

如果不考虑厂商造成的差异,Android 标准的分区还是能构成一个定义良好的文件系统层的。设备制造商以谷歌为 Android 模拟器提供的 Android 文件系统镜像为起点,对它进行改造,但通常也不会走得太远。接下来,我们将从 root (“/”) 开始,按照 mount 点的顺序讨论各个文件系统中存储的内容。

 注意,尽管文件系统是被存放在访问方式基本相同的存储器上的,但是你的 Android 设备上安装的各种硬件可以而且基本上都是五花八门的,所以很多厂商,包括谷歌自己,会额外为这些硬件提供一些专用的、被称为“proprietary blobs”的二进制可执行文件。每个设备所需的“proprietary blobs”(需要被提取到 AOSP 的/system 目录下的 device/子目录中)都会在一个名为“proprietary-blobs.txt”的文件中列出。

### root 文件系统

Android 的 root 文件系统 (/) 是 mount 自一个 RAM Disk (initramfs) 的。每次启动时,启动加载器 (fastboot) 从 boot 分区中把这个文件系统的镜像加载到内存 (RAM) 中,并把它交给内核。这一过程将在下一章中详细讨论,但就目前的讨论而言,最关键的一点是除非知道设备被“刷机”(flashed),否则 root 文件系统 (/) 是不能被修改的。这是很重要的,因为 root 文件系统中含有系统最重要的组件/init,它能以 root 权限执行任何操作,并控制着系统的启动过程。

一般 Linux 在启动过程中,通常是用 initramfs (以内核模块的形式) 把驱动加载到内核中,而且最终会在它被真正的文件系统取代后把它丢弃掉。不过在 Android 中却不是这样,Android 的 initramfs 还会驻留在内存中,并提供 root 文件系统 (/) 的功能。在实际操作中,这也仅限于提供表 2-5 中列出的/init 和其他几个配置文件和二进制可执行文件。

表 2-5 Android 的 root 文件系统 (/) 中存储的内容 (各 mount 点中的除外)

目录或文件	注 释
default.prop	其中记录的是编译时/build/core/main.mk 中 ADDITIONAL_DEFAULT_PROPERTIES 变量中的值,init 将根据它 (的内容) 去加载其他系统范围内的属性文件 (property)。加载只读的属性文件有利于强制执行安全保护
file_contexts	Kitkat: 记录 SE-Linux 中的文件 context。用于限制非授权用户访问系统文件和目录,将在第 8 章中讨论

续表

目录或文件	注 释
init	将会被内核以 PID 1 执行的二进制可执行文件，我们将在第 4 章中讨论它
init[...].rc	/init 的配置文件，主要的配置文件总是/init.rc，此外，一些设备或厂商也可以选择提供几个额外的配置文件，我们将在第 4 章中讨论这一问题
property_contexts	Kitkat: 记录 SE-Linux 中的系统属性（property）的 context。用于限制非授权用户访问系统属性，将在第 8 章中讨论
Seapp_contexts	Kitkat: 记录 SE-Linux 中的应用的 context。限制应用的操作域，将在第 8 章中讨论
Sepolicy	Kitkat: SELinux 策略设置编译后的结果（参见第 8 章）
sbin/	该目录中含有 adbd、healthd 以及（最重要的）recovery（即便不能 mount /system，系统也需要 recovery）等关键二进制可执行文件。该目录中可能还含有厂商提供的二进制可执行文件
verity_key	Android L 系统：含有认证/system 分区时所需的 DM_Verity RSA 密钥

## /system 分区

/system 分区是存放所有谷歌或厂商提供的 Android 组件的地方。该目录及其中存储的内容都是属于 root:root 的，所有的权限都是 0755（rwxr-xr-x），但是该文件系统是以只读方式 mount 的。这样做，主要是基于以下两个方面的考虑。

- **稳定性：**由于文件系统是以只读方式 mount 的，所以其中的内容也就不会受到破坏，即便是在设备突然断电时也是如此。这降低了因为该分区受到破坏而导致 Android 系统不能启动、使手机“变砖”的风险。
- **安全性：**以只读方式 mount 也是保护 Android 的系统组件不会被恶意修改的手段之一。尽管在真实情景中，只要以读-写参数重新 mount 一下这个分区就能搞定它了。

有些厂商，特别是 HTC，也会用闪存分区写保护（flash partition protection，HTC 称之为 S-OFF）的方式来保证 /system 是只读的。这意味着，即便用读-写参数重新 mount 了 /system，对它所做的任何修改也不会生效。从 KitKat 开始，谷歌开始不再对 /system 分区的完整性进行校验了，转而改用 Linux 内核中的 dm-verity 特性（详见第 8 章的讨论）保护 /system 分区的完整性。

对大多数设备而言，/system 分区中存储的（绝大部分）内容是一样的。尽管在理论上，其中的数据应该是完全一样的，但实际上，厂商和运营商有时也会向 /system 中添加一些它们自己的 App 和目录（加目录的情况比较少见），而不是把它们添加到专门为此设计的 /vendor 中去。表 2-6 中列出了你应该能在任意一台 Android 设备的 /system 分区中看到的内容。

表 2-6 /system 分区中存储的内容

目 录	注 释
app	存放系统应用：其中既有谷歌预先绑定的 App，也有厂商或运营商安装的 App（尽管从技术上说，这些 App 应该被放在/vendor/app 目录中，而不是这里）
bin	存放二进制可执行文件：其中既有多个守护进程的二进制可执行文件，也有一些命令行 shell 的二进制可执行文件（大多数指向 toolbox）
build.prop	在编译过程中生成的属性文件（property），/init 根据它（的内容）在启动过程中，去加载其他的属性文件
etc	其中存储了各种配置文件，它是/etc 的符号链接，表 2-13 中详细列出了其中的内容，请参考之
fonts	存放各种字体文件，也就是.ttf（True-Type Font）文件
framework	存放 Android 的框架（framework），各个 framework 会被存放在各自对应的.jar 文件中，每个 framework 中的 dex 代码在经过优化之后，会被存放在与.jar 文件同名的.odex 文件中
lib	存放运行时库文件，也就是原生的.so（ELF shared object）。在 Android 中，这个目录扮演了和正常 Linux 系统中的/lib 目录一样的角色
lost+found	在对/system 进行 fsck 操作时自动生成的目录。通常是空的（除非系统曾经崩溃过）。如果系统崩溃过，这个目录里可能会含有一些不知道自己的上一级目录是谁的 inode
media	以.ogg 格式存放了各个铃声、提示音、信息通知铃声、用户在操作图形界面时需要发出的各种声音，以及系统启动时播放的动画（详见第 5 章）
priv-app	这个目录中存放的是特权应用
usr	支持文件，比如 unicode 映射文件（icudt511.dat），设备和键盘的键盘布局文件等
vendor	专门用于存放厂商提供的文件（如果厂商有提供的话）。但厂商通常会把自己提供的内容存放到 /system 的各个子目录中（比如 bin/、lib/和 media/）
xbin	存放用途特殊的、正常操作过程中不需要的二进制可执行文件（不像 bin 中存放的那些都是正常操作中需要的二进制可执行文件）。在模拟器中，这个目录中通常存放的都是来自 AOSP 中 /system/extras 的各种工具。在实际销售的设备中，这个目录通常是空的，或者只含有 dexdump 这一个工具。各种提取 su 权限的 root 程序也会被放在这里

## /system/bin

/system/bin 目录中含有 Android 使用的各种原生可执行文件，此外，它也是存放各种调试工具的地方。具体来说，这些二进制可执行文件可以被分成 5 类。

- 用来提供服务的二进制可执行文件：这类二进制可执行文件都是在系统运行过程中由 /init 调用的，它们的调用路径会被写进/init 使用的 rc 文件中。这类二进制可执行文件并



非全都直接来自 AOSP，表 2-7 中标了灰色底纹的这些就是来自其他项目的。

表 2-7 /system/bin 中用来提供服务的二进制可执行文件

二进制可执行文件	功 能
app_process	用户 App 的宿主进程，Zygote（及其他所有用户 App）都是这个二进制可执行文件的实例，它是由 DalvikVM/ART 初始化的。
applypatch[_static]	在 OTA 升级的过程中使用——根据脚本来应用补丁，详见第 3 章的讨论。带后缀“_static”二进制可执行文件是静态链接版，相对于正常的（动态链接的）二进制可执行文件，在升级过程中，使用静态链接的二进制可执行文件可以减少对系统的依赖
bootanimation	当图形界面子系统（surfaceflinger）加载时，播放 Android 的启动动画。它常常会被制造商修改
clatd	IPv4-IPv6 地址的转换器
dalvikvm	用于启动一个 Dalvik 虚拟机的实例
debuggerd	在系统崩溃时产生 tombstone，也可以和一个远程 GDB 相连接
drmserver	第三方数字版权管理（DRM，Digital Rights Management）模块的宿主进程
dnsmasq	伪 DNS 服务：在设备对外提供 Wi-Fi 热点服务时，提供 DNS 代理服务
hostapd	Wi-Fi 热点守护进程的二进制可执行文件：在设备对外提供 Wi-Fi 热点服务时，提供一个模拟出来的 Wi-Fi 热点
keystore	Android 的密钥存储和管理服务
linker	Android 的运行时链接器。从本质上说，它并不是一个服务，但是在加载二进制可执行文件时，它却是必须的。要是它出了什么岔子，你的手机就铁定变成砖头了
mdnsd	组播 DNS（multicast DNS）的守护进程，用来使相邻的设备能通过和 Wi-Fi 直接相连的方式，相互发现和通信
mediaserver	音频/视频的录制/回放
mtpd	用以支持 PPP/L2TP
netd	用来管理网卡、防火墙等
pppd	点对点协议的守护进程，在使用 VPN 时需要使用它
racoon	提供对 VPN 支持
rlid	无线界面层（Radio Interface Layer）守护进程：主管一切与电话相关的服务
sdcard	SDCard 守护进程，实现了 SD 卡文件系统，通过 FUSE 模拟多用户权限设置（详见本章后面的讨论）
sensorservice	Sensor hub：并发读取各个传感器

续表

二进制可执行文件	功 能
servicemanager	提供了服务的注册和查找功能
surfaceflinger	它的功能是画出图形界面的样子，并把它们加载到 framebuffer 中
vold	Volume 守护进程：用来 mount/unmount 文件系统，也有文件系统的解密功能
uncrypt	解密文件系统（在 recovery 前使用）
wpa_supplicant	wpa_supplicant 适配层（wireless protected access supplicant）：提供 Wi-Fi 和 Wi-Fi P2P 的客户端支持

在第 5 章中讨论这些用来提供服务的二进制可执行文件是如何被/init 启动的，会对它们做更详细的描述。

- **调试工具：**被归入这一类的是一些用于调试的原生二进制可执行文件。表 2-8 中列出的这些二进制可执行文件在模拟器上都能找到，但是厂商（出于安全谨慎的考虑）可能会把它们从真实销售的设备中删掉。

表 2-8 /system/bin 中的调试工具

二进制可执行文件	功 能
adb	Android 调试桥（客户端）：这和你在宿主机上运行的那个 adb 实质上是一个东西（服务器程序位于/sbin/adbd）
asanwrapper	Address Sanitizer：内存溢出检测工具，它是个第三方调试工具
atrace	Android trace tool：使用 Linux 的 ftrace 来调试和跟踪程序的执行过程
bdt	BlueDroid（Android 中的蓝牙）测试 App
blkid	用来显示各个分区的 GUID
dex2oat	DEX 到 ART 的转换工具，把 DEX 文件中的代码编译成可以在当前设备上直接执行的 ART 格式。用以取代 dexopt
dexopt	DEX 优化工具。能够创建专为当前设备优化的 DEX 文件（在使用了 ART 的设备中就不再使用它了）
dumpstate	能与其他几个有用的工具（ps、dumpsys 等）一起使用的元工具（Meta-tool），用于抓取系统状态的调试快照
Dumpsys	服务转储工具：它能去连接 Android 的各个服务，请求调用服务中的 Dump()方法，以提供大量的调试信息

续表

二进制可执行文件	功 能
e2fsck fsck_msdos fsck.f2fs	分别对应 Ext2/3/4、vFAT 和 F2FS 文件系统的文件系统完整性检测工具。系统在 mount 这些文件系统之前，会自动运行它们
gdbserver	GDB 服务端：启动它之后，电脑上的 GDB 客户端就能通过 TCP/IP 协议连上当前设备，并调试相关的进程。大多数设备都会把它删掉
ip[6]tables	通过命令行，管理内核中的 IPTable（防火墙和流量限制）
keystore_cli	命令行实用程序，用于与 keystore（密钥存储）服务交互
logcat	把系统日志（/dev/log/*）打印到标准输出上去，它也支持一些过滤选项。这个命令非常有用，甚至直接可以以 adb logcat 的形式使用
ndc	与网络管理（Network Management）守护进程交互的命令行程序
perf	异常强大的性能分析（profiling）工具，它使用了内核中支持性能分析的一些功能
ping[6]	封包网络探针（Packet Internet Grouper），ICMP 协议的 echo 请求/响应包
radiooptions	用来模拟无线接口层（Radio Interface Layer）事件的测试程序
run-as	能用指定的 AID 权限运行某个应用程序
screencap	把帧缓存（framebuffer）抓取到标准输出或一个 PNG 文件中（供 ADB 使用）
screenrecord	屏幕录像工具（以.mp4 格式录屏）
screenshot	与 screencap 的作用类似，不过可以选择在截屏时发出声音
Service	与 service manager 交互的命令行实用程序
toolbox	Android 提供多种命令的二进制可执行文件，前文已述
vdc	与 vold（Volume 守护进程）交互的命令行实用程序
wpa_cli	与 wpa_supplicant 适配层交互的命令行实用程序

- **UNIX 命令：**为了让 shell 用户也能在 Android 上玩，UNIX 命令都被封装在一个单独的  
二进制可执行文件/system/bin/toolbox 中。busybox 是嵌入式系统中常见的多合一工具集，  
而 toolbox 则是它在 Android 中的定制版。toolbox 和 busybox 并不是逐条提供各个 UNIX  
命令的，它们只含有这些命令的基本功能，通过参数（比如"toolbox ls"）或者调用符号  
链接（如"ln -s /system/bin/toolbox /system/bin/ls"）的方式模拟执行相关命令。toolbox 中  
精简了 busybox 中的命令集，只提供其中的一部分常用命令<sup>1</sup>，但也加上了一些 Android

1 就像大多数定制 ROM 那样，在 Android 中也装上 busybox 是个不错的注意。busybox 中含有远比 toolbox 更为丰富的工具，这使得它对于高级用户来说是不可或缺的。——原注



特有的命令（如 `getprop/setprop/watchprop`）。

- **调用 Dalvik 的脚本（upcall script）：**这些调用 Dalvik 的脚本让用户通过 shell 与 Dalvik 运行时框架交互，这多半是为了进行调试。所有这些脚本（除了 `uiautomator` 外）的内容实际上都是差不多的，它们是从同一个母版上复制-粘贴过来，然后略做修改而得到的。这些脚本调用 `/system/bin/app_process`，用它们在 `/system/framework` 目录中的同名 JAR 框架，加载 Dalvik 类<sup>1</sup>，在使用时，脚本会把用户传给它的参数直接传递给 Dalvik 类。我们只需看一下代码清单 2-1 中给出的“am”脚本的内容，就基本等于了解了所有这类脚本的结构了。

代码清单 2-1 调用 Dalvik 的脚本模板

```
#!/system/bin/sh
#
# Script to start "am" on the device, which has a very rudimentary
# shell.
#
base=/system
export CLASSPATH=$base/framework/am.jar
exec app_process $base/bin com.android.commands.am.Am "$@"
```

表 2-9 中列出了这些脚本及其用途，如果不带任何参数地调用它们，将会输出一个简短的脚本用法说明。

表 2-9 /system/bin 中封装了 `app_process` 的脚本

脚 本	用 途
am	与 <code>ActivityManager</code> 进行交互、启动 activity 发起 intent 等
bmgr	备份管理（Backup Manager）接口
bu	启动备份
content	与 Android content provider 交互的接口
ime	控制输入法编辑器（Input-Method-Editors）
input	与 <code>InputManager</code> 进行交互，注入输入事件（在第 2 本中讨论）
media	控制当前的媒体播放器（播放/暂停等）
monkey	用随机生成的输入时间运行一个 APK

1 从 shell 中，用 `app_process` 启动 Dalvik 虚拟机，会比从 `Zygote`（它本身也是一个 `app_process` 的实例）fork 出一个虚拟机进程要慢得多。你只要运行一下上述任何一个脚本，就会发现这个问题。——原注

续表

脚 本	用 途
pm	与包管理器（PackageManager）进行交互，可以用来列出、安装、删除包（package）以及列出权限等
requestsync	同步账号（Sync accounts）
settings	获取、设置系统设置
svc	控制电源、数据、Wi-Fi 和 USB 服务
uiautomator	进行 UI 自动化测试，测试 dump view 之间的层次关系等
wm	与窗口管理器（WindowManager）进行交互、修改显示的尺寸/分辨率等

- 厂商定制的二进制可执行文件：从本质上讲，这类二进制可执行文件可以完全由厂商控制，但是这类二进制可执行文件通常都是些提供服务的程序或调试工具。Qualcomm（高通）提供了一套对于基于 msm 的设备通用的二进制可执行文件，见表 2-10。

表 2-10 /system/bin 中，Qualcomm 定制的二进制可执行文件

二进制可执行程序	功 能
mm-qcamera-daemon	高通内置摄像头的守护进程
mpdecision	Multi-Processor Decision：管理 CPU 主频的专用工具。它与 CPU governor 进行交互，能在系统忙时，加快主频或激活其他处理器（core）；也能在系统空闲时，降低主频或让一部分处理器休眠
qmuxd	高通基带访问多路转接器（baseband access multiplexer）
qseecomd	高通安全执行环境通信器（Secure Execution Environment Communicator）
thermal-engine-hh	thermal 守护进程，负责监视设备的温度，防止设备过热

除了芯片制造商之外 [高通、英伟达（NVIDIA）、OMAP 等]，这类二进制可执行文件还可能包括手机制造商（HTC、三星等）放入的一些其他二进制可执行程序，当然也可能没有。这些非 AOSP 的二进制可执行文件实际上是应该被放到别的地方去的，特别是放到/vendor 中。但是否遵从这一约定，则完全取决于各个厂商自身的意愿了。

 厂商提供的二进制可执行文件通常都是闭源的，而且很遗憾的是，在很多情况下，这些二进制可执行文件会在很大程度上影响系统的性能和安全，而且还可能会含有可资利用的漏洞。高通的 qseecomd 就是一个这方面的极好的例子。HTC 的 dmagent 也一样（它能被一个名为 WeakSauce 的漏洞利用程序利用，详见本书官方网站上的文章<sup>[5]</sup>）。



## /system/xbin

/system/xbin 目录类似于 UNIX 中的/sbin 目录,其中含有管理员会觉得非常有用的二进制可执行文件。但是普通用户最好还是离这个目录远一些。目录名中用的是“x”而不是原来的“s”,是为了避免与 Android 自身的/sbin(这个目录是 root 文件系统的一部分,其中含有系统操作时必须使用的二进制可执行文件)冲突。

这个目录中的二进制可执行文件是从 AOSP 的 system/extras 目录中编译得来的。因为这个目录不是普通操作所必须的,所以这个目录是不是会出现在设备中完全取决于厂商的意愿。事实上,有些厂商会干脆把这个目录删掉,或只保留一个 dexdump 二进制可执行文件。表 2-11 中列出的是在模拟器中,该目录里存放的额外的工具。

表 2-11 在模拟器的/system/xbin 目录中找到的 AOSP 二进制可执行文件

二进制可执行文件	功 能
add-property-tag	向系统的.prop 文件中添加属性
check-lost+found	在 fsck 操作完成之后,检查 lost+found 目录
cpueater	用于消耗 100% CPU 资源的死循环
cpustats	显示 CPU 和处理器调节器(用于控制主频)的统计信息
daemonize	把一个可执行文件作为守护进程启动,使之运行在后台,并关闭 stdin/stdout/stderr
dexdump	DEX 文件 dump 工具,也能 dump 文件头和字节码
directiotest	测试块设备的 I/O 性能
kexecload	使用 kexec 系统调用,用一个新的内核重写内核镜像
ksminfo	显示 KSM(Kernel Same-page Merger, 内核合并相同内存页)的使用信息
latencytop	以阅读起来更方便的形式,显示/proc/sys/kernel/latencytop 中的数据
librank	逐个给出各个共享内存区域在各个进程中的使用情况,主要给出 VSS/RSS/PSS/USS 信息
memtrack	(通过/proc/pid/smmaps)跟踪进程的内存使用情况的实用程序
micro_bench	内存基准测试工具
nc	Netcat, 分析 TCP 和 UDP 时的瑞士军刀
netperf netserver	网络性能分析工具(分别对应客户端和服务端)
procmem	显示进程内存使用情况统计信息(数据取自/proc/pid/status)
procrank	该程序的功能与 librank 互补,它会逐个进程地给出各个进程中 VSS/RSS/PSS/USS 的使用情况统计信息
rawbu	在底层备份/恢复 /data 中的数据





续表

二进制可执行文件	功 能
sane_schedstat	以人类可读的形式显示 scheduler 统计信息
showmap	显示进程内存分配情况（数据取自/proc/pid/maps）
showslab	显示内核 slab 分配器的信息（数据取自/proc/slabinfo）
SQLite3	命令行工具，由于 Android 中太多的 content provider 都支持 SQLite3，所以不论是在调试还是在电子取证工作中，这都是一个不可或缺的工具
strace	系统调用 trace 工具。使用 Linux ptrace(2)系统调用。强大的 trace 和逆向工程工具，往往能给你带来惊喜
su	切换用户为（root 或者其他用户）
taskstats	提供 Linux 的 taskstats 接口的详细使用统计信息
tcpdump	网络抓包工具。抓取到的数据文件可以用 Wireshark 打开
timeinfo	输出 realtime、uptime、系统处于唤醒状态时间的百分比和系统处于休眠状态时间的百分比

在真实的设备上，这些预先编译好的二进制可执行文件作为调试工具特别有用。把它们复制到设备里非常简单，只要使用 `adb pull /system/xbin` 命令，把模拟器中该目录中的文件全都复制到主机的某个目录中，然后再用 `adb push .. /system/xbin` 命令把它们全部写到设备中去即可（假设 `/system` 目录可写）。不论是 `/system/xbin` 还是 `/system/bin` 中的可执行文件，它们几乎都需要使用系统中的共享库才能正常工作。这些共享库可以在 `/system/lib` 目录中找到，详见下一节的讨论。

### /system/lib[64]

`/system/lib`（在 64 位系统中则是 `/system/lib64`）目录中含有供 `/system/bin` 和 `/system/xbin` 目录中的二进制可执行文件使用的共享库（在上一章中，我们大致介绍过其中的一些库）。在大多数设备中，`/system/lib` 中有多个子目录，其中的一些是根据设备的不同而不同的，但是下面这些子目录一般都会存在：

- `drm/`（提供 DRM 引擎。比如，实现了 forward-locking 专利代码包的 `libfwdlockengine.so`）
- `egl/`（Android 版的 OpenGL ES，详见第 2 本中的讨论）
- `hw/`（含有上一章在讨论过的各个 HAL 模块）
- `ssl/engines`（含有 `libkeystore.so`，该共享库使得 OpenSSL 能使用 Android 的 Keystore 机制）

在 Intel 设备中，`/system/lib` 通常还另外含有一个名为 `arm/` 的子目录，其中含有为 ARM 体系结构的处理器编译的共享库的拷贝。这些共享库会被用在 Intel 的二进制执行环境转换层（binary translation layer）Houdini 上，用来为要执行的 ARM 的二进制可执行文件提供一个完整



的（运行时）环境（因为在 APK 中常常会带有原生库）。

Nexus 设备中甚至还会含有更多的子目录，其中含有提供不同的 Google 服务（比如：Chrome/、Drive/、Wallet/等）的 JNI 库。

几乎所有的 Android 二进制可执行文件都是动态链接的。这一规则的一个例外是/sbin 目录中的二进制可执行文件（遵循传统的 UNIX 模型），这也就意味着，它们会在/system 分区没有被 mount 上来的情况下（所以当然也就没有/system/lib）被使用。下面这个实验将向你展示，你怎么才能准确地知道给定的二进制可执行文件需要哪些库。

### 实验：显示依赖给定库的所有二进制可执行文件，或者反过来玩

Android NDK 中有个非常重要的工具没有提供，它就是 ldd(1)。在 Linux 中，该工具是用来显示加载依赖关系的。Linux 版的 ldd 的工作原理实际上就是模拟加载一个二进制可执行文件，这也就是为什么在一台使用其他体系结构处理器的计算机上，用该工具分析依赖关系会失败的原因。在本书对应的网站上提供了一个名为“deps”的工具，它使你能像使用 ldd(1)那样，显示出一个可执行文件的所有依赖库。不过这个工具的工作原理是：扫描给定路径下的所有可执行文件，并把那些依赖于给定库的挑出来。

就像之前提到过的那样，当你在设备间移动二进制可执行文件，或者把二进制可执行文件从模拟器复制到真实的设备上去时，这个工具是非常有用的。模拟器中的/system/sbin 目录里的许多二进制可执行文件在真实设备上调试和跟踪时是极其有用的。把它们从模拟器中复制到真实设备中也不过是举手之劳（真实设备和模拟器当然应该用的是同一个 Android 版本），与此同时，当然也应该把所有缺失的依赖库也复制过去。例如，procrank 和 pagerank 都依赖于 libpagemap.so。deps 工具将会告诉你这一点（见输出结果 2-7）。

```
root@Generic:/ # deps -tree /system/sbin/procrank
procrank
+--libc.so ---- libdl.so
+--libm.so ---- libc.so
+--libpagemap.so +--libc.so
                  +--libm.so
+--libstdc++.so ----libc.so
```

输出结果 2-7 使用 deps 实用程序的一个例子

### /system/etc

就像它在 UNIX 系统中的同名目录一样，Android 的/system/etc 目录中也存放着各种配置文件之类的东西。/etc 也是该目录的符号链接，这样做是为了保持兼容性，因为有些 AOSP 外部项目会到这里去查找配置文件。表 2-12 中列出的是通常都能在这个目录中找到的一些东西。



表 2-12 一般能在/system/etc 中找到的文件和目录

名 称	描 述
NOTICE.html.gz	无数个 Android 开源组件的法律告知书，说明各种晦涩难懂的许可授权和法定事宜。由于这些文件不会被经常阅读（或者，你懂的……），所以它们是被放在一个经过 gzip 压缩的超链接文件中的
audio_effects.conf audio_policy.conf	会被 Android 的音频 HAL 使用（详见第 2 本书）
apns-conf.xml	Telephony provider 的配置文件，其中列出了设备所支持的所有运营商（它会被 com.android.providers.telephony.TelephonyProvider 使用）
asound.conf	设备 ALSA（Advanced Linux Sound Architecture）的配置文件，它会在某些设备上被使用
bluetooth/	BlueDroid 的配置文件
clatd.conf	CLATd（实现 IPv4 over IPv6）的配置文件
event-log-tags	各个不同 Android 系统组件的日志 tag（被 android.util.EventLog 使用）
fallback_fonts.xml	列出了在加载 system_fonts.xml 中没有指定的 font-family 时所能选用的 fallback font。它会被 Android 中的 layoutlib 的 FontLoader 所使用
gps.conf	GPS 配置文件
hosts	主机-IP 对应关系表，为了兼容起见，其中也记上了 localhost (127.0.0.1)
media_codecs.xml	列出了 StageFright 所支持的所有 codec（编码/译码器）（参见第 2 本书）
media_profiles.xml	列出了 LibMedia 所支持的所有 profile <sup>1</sup> （参见第 2 本书）
ppp/	存放了启动/停止 VPN 和 PPP 连接活动的二进制可执行文件
permissions/	存放了多个 XML 文件，每个 XML 文件规定了一个内置应用（不论是 AOSP 的还是厂商提供的）的权限，它会被 PackageManager 使用
security/	这个目录中含有存放着设备中硬编码写死的各个认证证书的目录（cacerts/）。OTA 升级的证书（otacerts.zip）以及经过签名验证的 APK 的 SELinux 标签（label），详见第 8 章
system_fonts.xml	按照字体（font）所属的 family 和 nameset，列出了系统字体（font），并把字体样式（font style）和/system/fonts 中的各个 TTF 文件一一对应起来，它会被 Android 中的 layoutlib 的 FontLoader 所使用
wifi/	WPA supplicant 适配层的配置目录，用于控制 Wi-Fi 和 Wi-Fi P2P 连接活动（见第 2 本书）

在不同的设备制造商（特别还要加上芯片制造商）手里，/system/etc 中可能还存有数量不确定的其他文件。表 2-13 显示了一般能在使用 MSM 芯片组的高通设备中找到的一些文件。

1 这个 profile 指视频的帧率、分辨率、编码质量等级等。——译者注





表 2-13 高通 (MSM) 设备的/system/etc 目录中的文件

名 称	描 述
*.acdb	各种声音校准数据库文件，在高通设备中会被 libacdbloader.so 使用
snd_msm/	存放高通 MSM SoC 音频设备的 ALSA 文件
thermal*.conf	监视设备温度的 thermald 守护进程的配置文件

## /data 分区

/data 分区是所有用户个人数据的存放地点。为此单独提供一个分区，有这么几个重要的好处：

- /data 与 Android 操作系统版本之间低耦合。系统升级和恢复时会擦除或者重写整个 /system 分区，但却不会以任何方式影响用户数据。反之，通过格式化 /data 分区，也可以快速重置设备，并擦除所有用户个人数据。实际上这就是在“恢复出厂设置”过程中所做的工作。
- 在用户需要时，/data 可以被加密起来。加密，不论怎么优化，由于在读写过程中分别要进行解密和加密操作，所以总会在一定程度上增加系统延迟。而在设计上，/system 分区中是没有敏感数据的，所以也就没有必要加密这个分区，因而也就规避了因加密这一分区而带来的延迟。
- /data 也可以被设为不可执行（即，用 noexec 参数 mount 该分区，或者强制执行 SELinux）。尽管到 KitKat 为止，这还不是默认选项。不过这样做，不光能使它更名副其实，还能极大地加大恶意软件的攻击难度。因为，这样恶意软件就没有一个可写又可执行的分区，让它能把恶意的可执行文件写在该分区中然后再执行了。因为 DEX 和 OAT 是运行在虚拟机里的，所以这对正常的 Dalvik/ART app 不会有任何影响。但是对 root 可能会有一定的影响（例如，需要重新 mount /data 以及 /system 分区）。

/data 分区是以 nosuid 权限 mount 的，这使得 root 设备的操作更加复杂了些。假设你已经以某种方式获取了 root 权限，但即使是在这个时候，二进制可执行文件 su（它提供了一个高效、持续的后门）还是必须要被放在 /system 这个只读分区中。事实上，这只是一个小小的障碍，因为以读-写模式重新 mount /system 只是举手之劳。不过不论如何，这也算是纵深防御的一个例子，而且在系统会对 /system 分区做密码学意义上的 Hash 校验时（就如同 KitKat 的 dm-verity 那样，详见第 8 章），这也被证明是一种确实有效的手段。

表 2-14 列出了存储在 /data 分区中的内容，不过注意制造商和销售商可能还会在其中再加上



的一些其他的文件或目录。


表 2-14 /data 分区中的目录

目 录	注 释
anr	dumpstate 用来记录失去响应的 Android 应用的函数调用栈当前状态的地方。根据 dalvik.vm.stack-trace-file 属性的默认设置，函数调用栈的当前状态会被记录在 traces.txt 文件中
app	用户自己安装的应用，下载下来的.apk 文件都可以在这里被找到
app-asec	存放 asec 容器，当一个应用使用了 asec 保护技术时，它就会被加密起来，然后放入一个 asec 容器，每个应用占用一个 asec 容器（详见本章后面的描述）
app-lib	应用（不论是系统应用还是用户自己安装的应用）的 JNI 库都可以在这里被找到
app-private	提供应用私有存储空间（application private storage），不过在实践中已经很少使用了，因为 asec 提供了更高的安全性
backup	供备份（backup）服务使用
bugreports	bugreport 专用，用来存放 bugreport 生成的报告，每份报告中均含有一个文本文件和一张屏幕截图（png），这两个文件均以 bugreport-yyyy-mm-dd-hh-mm-ss 的格式命名
dalvik-cache	用于存放优化过的系统应用和用户安装的应用的 classes.dex。每个应用的 dex 文件名都是它 apk 包的存放路径，并用“@”替换掉了路径分隔符（比如，system@framework@bu.jar@classes.dex）
data	各个已安装应用的数据目录，目录名为逆 DNS 格式，详见后面的讨论
dontpanic	原本打算用来存储 Android 的 panic console 和 thread。现未使用
drm	供 Android 的数字版权管理器（Digital Rights Management）使用
local	供 uid shell 使用的一个可读/可写的临时目录（也可在 ADB 会话中使用）
lost+found	对/data 分区执行 fsck 操作时自动生成的目录。一般为空（除非文件系统崩溃。在这种情况下，其中可能含有未知上级目录的 inode）
media	供 sdcard 服务把 SD 卡 mount 到这个 mount 点上
mediadrm	供 Media DRM 服务使用
misc	供各个组件存放“各式各样的”数据和配置文件的目录，参见表 2-17
nfc	存储 NFC 参数
property	存放持久性属性（persistent properties，即设备重启后仍被保留下来的属性）。每个属性都被保存在它自己的文件中，文件名就是属性名
resource-cache	供 AssetManager 使用的资源缓存（详见第 2 本书）
security	通常为空
ssh	供那些提供 ssh（Secure Shell）服务的设备使用（一般为空）



续表

目 录	注 释
system	存放了大量的系统配置文件，详见表 2-18
tombstones	用于存放由 debuggerd 生成的应用崩溃报告。受文件系统空间大小的限制，存放完整的 core dump 是不可行的。debuggerd 在不能提供 core dump 时，会提供基本的 autopsy 服务。有些厂商会为该目录分配一个独立的空间
user	Jelly Bean 及之后的版本引入。不同的用户会把各自的数据和应用存储/安装在/data/user/用户号/（用户号从 0 开始顺序编号，0,1,...）下的各个目录中，系统运行时，把/data/data 下的对应目录做符号链接，使之指向/data/user/用户号/下的对应目录，以这种方式让 Android 系统能支持“多用户”。在一个单用户系统中，/data/data 会被直接指向/data/user/0

 /data 目录的权限和/data/data 的一样，都被设为 chmod 771 system system。这一做法源于 Android 安全模型的要求：对于所有的应用而言，目录都是可执行的（即可以用 cd 命令切换），但同时又是不可读的（这样，应用或者不受信任的进程就不能将“相邻”目录一一列出）。这也就意味着，在 uid shell 里（也就是没有 root 权限的 adb 会话中），你可以把当前目录切换到/data 目录或者它的大多数子目录上，但没有必要让你能读取其中的内容。system 子目录及其子目录（即/data/system 和/data/misc）是可读的，但是/data/data 和/data 本身是拒绝执行 ls 命令的。从 KitKat 起，这一做法通过 SELinux 标签被引入，你要想遍历各个子目录就得有 root 权限才行。

/data/data

/data/data 这个名字看上去有些罗嗦的目录是所有应用（不论是系统应用还是用户安装的应用）存储它们的信息的地方。每个应用都有一个单独的、以逆 DNS 格式命名的子目录。其权限设置为 chmod 751 (rwxr-x--x)，前二者为应用拥有者的 uid 和 gid 的权限。/data/data 目录本身的权限设置为 chmod 771 system system，这使得所有的应用都能列出其中的子目录，但却不能读取除了 system 拥有的应用外的其他数据。不过应用中，文件的安全防护措施的设置还是要交给每个应用自己来完成，因为尽管除了应用的拥有者外，其他人都不能读取各个应用目录中的内容，但是这些应用目录本身还是可执行的。

/data/data 下各个应用的子目录是各个应用在整个文件系统中唯一能写入数据的地方。再加上很多市面上能找到的，而且也会被装到很多手机里去的应用都能定位 GPS 位置，收发短信和打电话，这就使得这个目录中的一些地方能够成为电子取证时的“聚宝盆”。我把我们特别关心的一些子目录列在了表 2-15 中。



表 2-15 /data/data 中一些有意思的应用目录

App 子目录	使用者	内 容
com.android.providers.calendar	Calendar	日历数据库 databases/calendar.db（其中记录的是事件列表中的信息）
com.android.providers.contacts	Phone Contacts	事实上，databases/contacts2.db 中存放的每条记录都会引起调查人员的高度注意。在这个 SQLite3 格式的联系人数据库中，存放了 contacts（其中记录了存储在设备上的所有联系人的信息）和 calls（其中存储了最近的通话记录）等多张表。每个联系人还会有由自己的简介（files）和头像（thumbnail_photo_XXXXX.png）组成的 thumbnail
com.android.providers.telephony	Messaging	彩信（MMS）/短信（SMS）数据库 database/mmssms.db
com.android.providers.settings	Settings	databases/settings.db：所有 Android 框架的运行时设置，在 global 和 secure 表中还记录有更多的信息
com.google.android. apps.maps	Google Maps	查找目的地：在 gmm_myplaces.db、gmm_storage.db 和 log_events.db。cache/http 中记录了地图 tile <sup>1</sup> 的信息
com.google.android.gm	GMail	databases/mailstore.email.db：这个 SQLite3 数据库中存储了用户所有已经注册过的邮箱（这些信息存储在 messages 表中）里那些已经被下载到该设备上的电子邮件。已经看过的附件被存放在 cache/email 数据库中
com.android.chrome	Chrome Browser	Chrome 浏览器（它替换了旧版本 Android 内置的 com.android.browser）的地盘。其中我们感兴趣的包括：cache/目录（浏览器的缓存）和 app_chrome/Default/目录——在该目录中存放了许多重要的 SQLite3 数据库，比如 History 和 Archived History（上网历史记录存放在名为“urls”的表中），Login Data（在 logins 表中存储了各个网站的登录用户名-密码）以及 Cookies

应用也可以把数据保存在 SD 卡上（如果它们有权限的话），但是大多数与应用状态相关的数据一般还是能在/data/data 中的对应目录里找到的。如果你想手动保存和恢复应用的状态的话（比如，在游戏中作弊），这是非常有用的。应用也可以注册 Android 的备份服务，自动进行备份（备份可以放在本地，也可以放在谷歌的云服务器上），这将在下一章中予以讨论。

表 2-15 列出的当然远远不及复杂的实际情况的万分之一。不过，如果你对某个特定应用中的文件感兴趣，你可以直接去/data/data 目录，根据逆 DNS 格式的目录名（实际上就是 APK 的名称）找到这个应用。然后，（在一台已经 root 了的设备上）可以很方便地直接把其中的文件

1 地图 tile 指的是打开地图时（特别是网速慢时）逐渐显示的一个一个小方块，一个小方块称为一个 tile。——译者注

全部复制出来,然后再用 SQLite3 查看各个数据库中存储的内容,或者如果该文件不是 SQLite3 数据库的话,你也先用 file 命令识别其他文件的类型,然后再用相应的工具查看其中的内容。就像我们在下面这个实验中所做的那样。

### 实验:通过分析/data/data 中的数据对移动设备做电子取证

在一个已经 root 了的设备上,你可以很方便地用 SQLite3 检查应用的数据目录。Android 模拟器镜像中的/system/xbin 目录里含有一个名为“sqlite3”的二进制可执行文件,而且大多数的 root 工具包里也有这个可执行文件(这么做的理由现在看已经相当明显了)。

我们以 Chrome 浏览器为例。启动浏览器,随便浏览一个网站,为了锁定数据库的状态,让你看到历史记录数据库的原始状况,你需要杀掉浏览器进程。现在只要一条 SQL 查询语句就全搞定了(见输出结果 2-8)。

```
root@htc_m8wl:/ # cd /data/data/com.android.chrome
# Using ".schema" shows the table definition:

root@htc_m8wl:/data/data/com.android.chrome # _sqlite3_app_chrome/Default/History
sqlite> .schema urls
CREATE TABLE urls(id INTEGER PRIMARY KEY,url LONGVARCHAR,title LONGVARCHAR,
visit_count INTEGER DEFAULT 0 NOT NULL,typed_count INTEGER DEFAULT 0 NOT NULL,
last_visit_time INTEGER NOT NULL,hidden INTEGER DEFAULT 0 NOT NULL,
favicon_id INTEGER DEFAULT 0 NOT NULL);
CREATE INDEX urls_url_index ON urls (url);
sqlite> select * from urls where url like "%android%";
id|url|title|last_visit_time|
52|http://newandroidbook.com/|Android Internals|2|2|13054934895637919|0|0
53|http://newandroidbook.com/T|mailto:edward.suhko@gmail.com|Internals::TOC|1|0|13054934883061164|0|0
```

输出结果 2-8 用 sqlite3 检查 Chrome 浏览器的历史记录

还是用这一招,我们还能检查/data/data/com.android.providers.contacts/databases 目录中的 contacts2.db 数据库(见输出结果 2-9)。

```
sqlite> .schema calls
CREATE TABLE calls ( id INTEGER PRIMARY KEY AUTOINCREMENT,number TEXT,
presentation INTEGER NOT NULL DEFAULT 1,date INTEGER,duration INTEGER,
...
# E.g. find all toll free calls
sqlite> select id, number, date, duration from calls where number like "%800%";
id|number|date|duration
2|18001750930|1396019679278|0
16|18007562000|1402005179460|0
```

输出结果 2-9 查看通话记录

另一个有用的电子取证技巧(这一招甚至不需要设备已经解锁或 root)是通过 adb 把设备与一台电脑连接起来,对我们所关心的包发出一条 adb backup 请求。这会调用

BackupManagerService，由于它是以 system 权限运行的，所以能够毫无限制地访问/data/data 目录，读取任何一个应用中的所有文件，还能很方便地把数据传送到电脑上（备份过程和 BackupManagerService 分别会在下一章和第 2 本中详细讨论）。

在备份操作启动之初，BackupManagerService 会提示用户进行确认（因此也就需要一台已经解锁了的设备）。如果用户同意进行备份，在电脑上就会创建一个扩展名为.ab( Android Backup 的缩写)的备份文档。当你了解了这种文件的格式之后（这将在下一章中予以介绍），备份的数据就能很方便地被提取出来。

/data/misc

/data/misc 目录中含有各个 Android 子系统的各种五花八门的（miscellaneous）数据和配置目录。和它的名字所暗示的相反，该目录中也含有一些系统中最重要文件。更多信息详见表 2-16。

表 2-16 /data/misc 中的目录

目 录	内 容
adb	存储可信的允许进行 ADB 连接的电脑的公钥（Jelly Bean 及之后的版本）
bluetooth	BlueZ [<4.2 的 bluetooth（蓝牙）子系统] 的配置文件
bluedroid	Bluetooth（蓝牙）子系统（>4.2）的配置文件
dhcpc	存储实现 dhcp 的 ctdent 守护进程的 PID 文件，以及所有当前活跃的地址租用记录
keychain	存放 Android 内置证书 pin 码（certificate pin）和黑名单
keystore	存储每个用户的 keystore 数据
sensors	用于存储传感器调试数据
sms	存储短信（sms）codes 数据库
systemkeys	用来存储打开 ASEC 容器的密钥（AppsOnSD.sks）
vpn	用于存储 VPN 状态配置文件
wifi	用于存储 Wi-Fi 子系统的配置文件（比如：wpa_supplicant.conf）和套接字（socket）

/data/system

/data 分区中另一个重要的子目录是/data/system，因为该目录中含有对维护设备状态非常重要的文件。你都应该能猜到了，这个目录的访问权限也是被限制为 system:system 的。所以如果你的设备没有被 root，你是看不到表 2-17 中给出的任何一个文件的。



表 2-17 /data/system 中的内容

目 录	注 释
appops.xml	供控制应用权限的 AppOps 服务使用
batterystats.bin	供统计各个应用电量使用情况的 BatteryStats 服务使用
called_pre_boots.dat	供 ActivityManager 记录每个 boot broadcast receiver
device_policies.xml	DevicePolicyManagerService 使用的配置文件
dropbox/	供 DropBox 服务使用的目录
entropy.dat	系统熵存储器 (system entropy store), 供 EntropyMixer 生成随机数使用
gesture.key	锁屏图案的 Hash, 详见第 8 章讨论
framework_atlas.config	供负责将预加载的 bitmap 组装成纹理贴图的 AssetAtlasService 使用
ifw/	Intent 防火墙规则库 (参见第 8 章)
locksettings.db*	锁屏设置: 其中记录了设备的锁屏策略 (参见第 8 章)
netpolicy.xml	供 NetworkPolicyManagerService 使用的配置文件
netstats/	用来记录 NetworkStatsService 按 device、uid 或 xt 得到的网络传输数据统计的目录。之前版本的 Android 中只是把这些文件都放在/data/system 目录中
packages.list	PackageManager 列出的所有安装在系统中的包 (APK)
packages.xml	PackageManager 在此记录了所有已安装包的元数据
password.key	锁屏 PIN 码/口令的 hash, 详见第 8 章讨论
procstats/	供 ProcessStats 服务存储文件的目录
registered_services/	供 android.content.pm.RegisteredServicesCache 使用的目录
usagestats/	供 UsageStats 服务存储文件, 特别是 usage-history.xml 文件的目录
users/	Android 的“多用户”支持, 详见第 8 章讨论

## /cache 分区

Android 是在系统升级的过程中使用/cache 分区的。系统升级包会被下载到这里, 启动管理器 (boot manager) 特别是在 recovery/升级模式下启动时, 会要使用这个分区。但除此之外, 在正常情况下, 这个分区是空的。

如果你最近下载过 OTA 升级包, 在它被安装之前, 你都能在这个分区里看到它。另外, recovery 这个二进制可执行文件和系统 (特别是 android.os.RecoverySystem 类) 在启动到 recovery (或系统升级) 模式时, 也会使用这个分区交换信息, 如表 2-18 所示。

表 2-18 /cache 分区中的各个（子）目录

recovery 中的宏定义	路 径	用 途
CACHE_LOG_DIR	/cache/recovery	二进制可执行文件 recovery 的专用目录
LAST_LOG_FILE	/cache/recovery/last_log	上一次 recovery/升级操作的日志
LOG_FILE	/cache/recovery/log	当前 recovery/升级操作的日志
COMMAND_FILE	/cache/recovery/command	传递给 recovery 的命令行参数
INTENT_FILE	/cache/recovery/intent	recovery 完成之后要发出的 intent
LAST_INSTALL_FILE	/cache/recovery/last_install	最近一次安装日志
LAST_LOCALE_FILE	/cache/recovery/last_locale	存放再次启动时使用的语言设置

recovery 和系统升级的详细过程，见第 3 章讨论。

## /vendor 目录

/vendor 目录是用来存储厂商对 Android 系统的修改的。这样做是为了在必要时能有效地进行系统的更新或升级操作（同时不至于把厂商对系统的修改一起抹除）。专门指定的系统组件在添加/system 路径之前时，会先去检查事先被写死在程序中的/vendor 中的路径，具体哪些组件会去检查哪些路径，详见表 2-19。

表 2-19 各个系统组件会去搜索的/vendor 中的路径

组件名	将会去搜索的路径
Package Manager	/vendor/app
Fonts	/vendor/etc/fallback_fonts.xml
Shared Libraries	/vendor/lib
DRM libraries	/vendor/lib/drm /vendor/lib/mediadrm
eGL libraries	/vendor/lib/egl
Frameworks	/vendor/overlay/framework
Firmware	/vendor/firmware
Audio Effects	/vendor/etc/audio_effects.conf

不同设备的/vendor 中存储的内容会有极大的差别，因为各个厂商会根据他们自己的意愿添加自己的应用和组件。有些厂商，比如亚马逊（Amazon），会创建他们自己的子目录（比如 /vendor/amazon），用以存放定制的框架和特性（比如，Kindle 根据 CSV 文件调整音量的子目录，以适应不同的输出设备的“smart volume”特性，就被存放在/vendor/amazon/smartvolume 这个

路径里)。另一些厂商则会完全忽略掉这个目录,而是把他们对系统的修改直接放在/system 分区中。这一做法在厂商预装应用时最为常见,事实上/vendor/app 目录基本上是没人去用的(即便是在亚马逊的 FireOS 中也是如此)。这使得删除厂商预装的应用和厂商额外提供的很占空间的软件非常困难。如果 Nexus 9 上使用的 Android L 系统对 Android 系统的进一步改进有什么暗示的话,或许,今后版本的 Android 系统会提供一个单独的分区给/vendor,这使得厂商对系统的修改能与系统的其他部分相互剥离,能够单独地对它进行升级。

## SD 卡

Android 最棒的一个特性就是:它内置支持 SD 卡。这是许多 iOS 用户梦寐以求的东西<sup>1</sup>。现在大多数手机已经都自带一个 SD 卡插口了(尽管插-拔都不是很方便),平板电脑上也已经有了插-拔都很方便的扩展插口了。

大多数 SD 卡都会被格式化成 vFAT 或者 FAT32 文件系统,但是这个类型的文件系统是不支持权限的。为了能强制实现权限,以及支持(从 JellyBean 开始提供的)多用户配置,Android 通过 FUSE(用户态下的文件系统)模拟 SD 卡的方式,算是勉强解决了这个问题。FUSE 使我们能在一个用户态[user mode,这也是 FUSE 的名称后半部分的由来,前面的 F 表示文件系统(File system)]进程中,而不需要在内核中实现文件系统。FUSE 会在内核中安装一个用于支持通用文件系统的小模块,通过它和 VFS 对接并实现基本的注册文件系统功能,而真正的文件系统是在一个用户态进程/system/bin/sdcard 中实现的。在 Android 的演化过程中,SD 卡的 mount 点被改过好几次,现在最新版的 Android 中,该 mount 点在/storage/ext\_sd。对于没有 SD 卡的设备,这个 mount 点通常会指向/data 分区中的某个目录(它一般是/data/media/0),在输出结果 2-10 中显示了该系统的 SD 卡 mount 点及 SD 卡默认的目录结构。

上面这些标准目录已经在 android.os.Environment 类中被定义为常量了。请注意,第三方应用也能(而且经常会)在 SD 卡中创建它们自己的文件和目录。

Android 还提供了一个模拟 SD 卡文件系统,可以在设备中没有 SD 卡时使用,也可以和“真正的”SD 卡文件系统同时使用。使用 mount 命令,你可以观察到 SD 卡文件系统(见输出结果 2-11)。

---

1 实际上,iOS 本身确实也是支持 SD 卡的,但插入 SD 卡的唯一办法是使用“相机连接套件”(Camera Connection Kit)——实际上,这玩意就是个 USB 转换器。当然,这需要果粉们再花上 29.95 美元(或者更多的钱),而且用的时候还会占掉设备上唯一的数据-充电线插口,要是用户同时还想充电的话,可就抓瞎了。——原注



```
shell@android:/ $ cd /mnt/sdcard
shell@android:/mnt/sdcard $ ls -F
Alarms/
Android/
DCIM/           # Shared with host when connected as camera
Documents/
Download/
Movies/
Music/
Notifications/
Pictures/
Playlists/
Podcasts/
Ringtones/
```

输出结果 2-10 SD 卡的目录结构

```
shell@htc_m8w1:/ $ mount | grep fuse
/dev/fuse /mnt/shell/emulated fuse rw,nosuid,nodev,relatime,user_id=1023,group_id=1023,...
/dev/fuse /storage/ext_sd fuse rw,nosuid,nodev,relatime,user_id=1023,group_id=1023,...
```

输出结果 2-11 观察 SD 卡文件系统

第 5 章中将侧重于从 sdcard 守护进程的技术角度，开展更进一步的讨论。

## 2.3 受保护的文件系统

Android 是个开放系统这一事实，使得应用在部署时会遇到一点麻烦。一个稍微懂点技术的精明用户就能使用 adb 在设备间复制 apk 包。要是应用是被安装在 SD 卡里的话，由于 SD 卡本来就可以拔下来，插在其他设备上，就使得即便一个菜鸟也能把装在一个设备里的应用复制到另一个设备中去。不像 iOS 系统在设计之初就考虑要支持 DRM（使用它的 FairPlay 机制，并默认加密应用的代码），Android 是在不断的版本更新过程中慢慢地开始使用一些机制来达到同样的目的，本节中将会讨论两种这类机制——OBB（Opaque Binary Blob）和 ASec（Android 安全存储，Android Secure Storage）。前者使得攻击者在攻击时需要绕过 APK 中的限制，而后者是专为保护应用不被复制到其他设备中去而设计的。

### OBB: Opaque Binary Blobs

谷歌商店把 APK 包的大小限制在 50MB 以内。对于某些应用（特别是那些运行时需要使用或者处理多媒体文件的应用）来说，这是个比较棘手的限制。从 GingerBread（Android 2.3）开始，Android 开始支持 OBB（Opaque Binary Blob）格式，这使得开发人员可以以 Opaque Binary Blob 或者简称 OBB 文件（可以把多个多媒体文件存放在一个的 OBB 文件中，而且还可以选择对数据进行加密）的形式，为应用提供最多 2GB 的额外的数据文件。

要说漂亮的实现，OBB 绝对算是一个。OBB 中的“O”，代表“Opaque”，不透明。这意味着，其中存放的数据以及使用的格式将由应用的开发者来决定。尽管在许多时候，它就是个 vFAT 文件系统镜像，由 volume 守护进程来 mount 它。vold 随后将会调用 Linux 内核中的 device mapper，用 loop 参数执行一次 mount 操作<sup>1</sup>。device mapper 也支持 two-fish 加密算法，在发起 OBB mount 请求时，可以把密钥一并传递给它。应用程序可以调用 android.os.StorageManager 中的 mountOBB 方法，来用指定的密钥 mount OBB 文件。这一过程如图 2-1 所示。



图 2-1 OBB 文件的 mount 过程

尽管会尽量做到不透明，但是 OBB 文件中仍然需要使用某些类型的元数据，以便系统能够解析它。支持 OBB 的代码位于原生库/system/lib/libandroidfw.so 中，我们看一下编译生成它的 ObbFile.cpp 文件，就会发现 OBB 文件的元数据位于文件尾部（而不是我们一般认为的那样：位于文件的头部）。因此解析 OBB 时，也需要先跳转到文件的尾部，然后倒着往前解析。文件尾部的元数据中各个字段的含义如图 2-2 所示。

Android 源码树中含有一个 obbtool 工具——这是个可以用来在 Linux 中创建 OBB 文件的 shell 脚本——它首先会创建一个空的 vFAT 镜像，然后使用电脑上的 device mapper 以 loop 参数 mount 它。mount 了之后，就可以往里面添加文件了。在 unmount 时，相关数据会被写入镜像。在 SDK 中还提供了一个能创建和维护 OBB 文件的 jobb 工具<sup>[6]</sup>。在 Android 开发框架中也有一个 ObbScanner 类，用它能获取 OBB 文件中基本的元数据（通过 JNI 函数调用上面提到过的 libandroidfw.so 中的函数）。在 APK Expansion Files<sup>[7]</sup>网页上有讨论 OBB 的详细文档，你可以像第 5 章中将要讨论的那样，用该 vdc 命令与 vold 守护进程交互，列出 mount 和 unmount OBB 文件。

<sup>1</sup> 原文 loop mount 不是说循环 mount 某个东西，而是指用 loop 参数 mount。也就是说，把文件系统中的一个文件视为一个新的块设备（文件系统）予以加载的情况，这里的 loop 是指文件系统又嵌套有一个文件系统的情况。在其他一些文献里又称之为“回环设备”。——译者注

Signature Version	OBB包所属的版本号
Package Version	版本号（当前只有一个版本）
一些标志位	当前没有定义任何标志位
64位的salt	
包名的长度	strlen(包名)——最小值是1
...	包的名称——至少要有1个字节
Footer Size	
总是32+strlen(包名)	
0x01	0x05
0x99	0x83
文件签名	

图 2-2 OBB 文件尾部的数据格式

## A Sec: Android 安全存储（Android Secure Storage）

Android 的“安全存储”（Secure Storage）特性，通常也被称为 asec，它提供了一种让应用能安全地安装到设备上的机制——它确保在一个“合理的前提假设下”，用户没法把（安装在一台设备上的）应用拷贝到另一台设备上去——该过程通常被称为“预先锁定”（forward-locking），通过使用 asec 容器（container），应用可以被安装到任何地方。该特性从 Android 2.2/2.2.1 版（Froyo）起被添加进了 Android，Froyo 是第一个支持 SD 卡之类的扩展存储的 Android 版本。实际上，asec 容器也可以被存放在 SD 卡里，但是没有密钥就没法使用。显然，密钥是需要被存放到其他什么地方的——Android 把它们放在系统密钥存储器（keystore，位于 /data/system/misc/systemkeys）中。因此，这个“合理的前提假设”就是 root 用户是能够读取加密密钥的。

实际上，asec 容器就是一个由固定的文件头 asec\_superblock 起始的加密过的文件系统镜像。asec\_superblock 的定义位于 system/vold/asec.h 文件中，其格式如图 2-3 所示。

asec 的创建和管理都是由卷管理器（volume manager）vold 完成的。vold 根据 MountService 发来的指令，执行相关操作。asec 的创建和 mount 过程都需要一个密钥，在 asec 容器被 mount 的时候，vold 会使用内核中的 device mapper，用 loop 参数执行 mount 操作，并通过 DM\_TABLE\_LOAD ioctl 操作，把密钥传递给内核中的 dm-crypt 模块。

你可以使用 df 命令来查看已经被 mount 上来了的 asec，也可以使用 vold 的命令行工具 vdc 列出详细信息（详见第 5 章）（见输出结果 2-12）。




0xC0	0xDE	0xF0	0x0D	幻数 (文件签名)
1	版本号 (当前就只有一个版本)			
c_cipher	加密算法: 0-不使用, 1-TwoFish, 2-AES			
c_chain	Chaining (unused - currently only: 0 - None)			
c_opts	Options: 0 - None, 1 - Ext4			
c_mode	Mode (unused - currently only: 0 - None)			

图 2-3 asec 文件头 (取自 system/vold/asec.h)

```
shell@4$ df | grep asec
/mnt/asec          930.8M    0.0K  930.8M  4096
/mnt/asec/com.tripadvisor.android.apps.cityguide.shanghai-1 23.0M   21.3M   1.7M   409
#
# Use the Volume Daemon Command utility (vdc) to examine mounted asec
#
shell@4$ vdc asec list
111 0 com.tripadvisor.android.apps.cityguide.shanghai-1
200 0 asec operation succeeded
```

输出结果 2-12 观察 asec 文件系统

加密理论中有个古已有之的问题——密钥的管理。换言之，我们该把 asec 容器的加密密钥存放在哪里呢？如果要加密密钥本身，你马上就会陷入“先有鸡还是先有蛋”的问题。所以 Android 选择把这个（128 位的 BlowFish）密钥单独存放在/data/misc/systemkeys/ AppsOnSD.sks 文件中。该文件以明文形式存放着密钥，但是被设置为只有 root 才能读它。显然，这意味着，如果在一台已经被 root 了的设备中试图使用 asec 保护知识产权，只能是竹篮打水一场空。

 如果读者您对亲自动手操作 asec 容器感兴趣的话，*Android Explorations blog post about JB's App Encryption*<sup>[8]</sup>将是个不错的参考文档。

如果你认为 asec 特性与上面讨论的 OBB 很像的话，这绝对不是巧合。这两种特性都使用了 device mapper 和它的文件加密功能(dm-crypt)创建和访问数据。asec 实际上可以被视为 OBB 的逻辑升级版，从加密存储应用的扩展文件，升级到加密存储整个应用。该机制同样可以扩展到整个文件系统级别上，事实上这就是 Android 用来加密整个磁盘的特性，详见第 8 章的讨论。

## 2.4 Linux 伪文件系统

尽管不是严格意义上的 Android 文件系统，但 Linux 内核中提供的其他三种值得注意的文件系统也在 Android 上被使用。这些文件系统对我们第 7 章中的讨论特别重要，在第 7 章中（除了其他的一些问题之外）讨论了 Linux 视角下的 App，即，从 Android App 在 Linux 系统层级中的表现形式——进程——这一角度跟踪（trace）和分析它们。这一节中，我无意于详细讨论这些伪文件系统的所有目录，只是阐述一下与之后的讨论息息相关的那些重要的路径。

请注意术语“伪文件系统”：这些文件系统没有一个会被存储到物理存储设备上去的，相反它们是直接由内核中的回调函数维护的。也就是说，当要访问其中的一个文件或目录时，某个对应的内核级处理函数就会被调用。这也意味着，这些文件系统并不真正占用存储空间（在内核所用的内存中，用于存放对应的 inode 和目录数据结构的开销不算）。更进一步说，每次访问伪文件系统中的一个文件或目录都要去调用系统的回调函数，所以这些文件和目录反映出的总是最新的实时更新的数据。由此可以推出，讨论（伪文件系统中）文件的大小是没有意义的，这也就是为什么用“ls -l”命令看到的这些文件都是空的 [或者在旧版本的内核里，是 4KB（即一个内存页大小）的整数倍] 的原因。注意，由于这些文件都是由内核代码（由整个内核，或者在某些情况下，由某些内核模块）导出的，所以根据内核版本的不同，这些文件有时也会有所不同，而且文件中的内容（特别是 sysfs 中文件的内容）也是和硬件紧密相关的。

大多数伪文件系统中的文件都是以只读权限创建的，因为其用途是提供实时的诊断信息，向用户态程序提供一种使之能够查看一些内核态中原本是不可访问的变量和结构的机制。不过有些文件实际上是可写的，这提供了一种更有用的能力，使得我们在用户态中就能实时地对内核中的数据施加影响。不像在某些基于注册表的系统中，修改配置需要对各种隐藏的而且通常是不公开文档的注册表键和值进行操作（更别说还要重启）那样，对伪文件系统中（允许修改的）文件进行的修改，会立即产生作用，而且还不需要系统重启。不过这一特性并没有太多人注意，因为在系统重启时，再重新进行一遍同样的配置，实在是太烦人了。事实上这也就是 Android 中的 init.rc 脚本的重要功能之一（详见第 4 章讨论）。

### cgroups

Linux 内核提供了一种重要的资源控制机制，它就是 cgroups。一个 cgroup 就是一个可容纳一个或多个线程的组容器（container group），它能把组中的所有线程视为一个整体，统一进行操作和策略设置。在 Linux 内核文档<sup>[9]</sup>中，提供了关于 cgroups 的相当详尽的文档。为了便于在用户态中能把各个线程放置到不同的组中，cgroups 通过伪文件系统把它们自己导出到用户态中——这使得只需对这些文件进行简单的“写”操作，就能把一个线程添加到某个组中。

尽管 cgroups 用途广泛，可以以各种不同方式使用，但是在 Android 中却是以一种非常受限的方式使用它的——只用于 CPU 使用记时和线程调度（见输出结果 2-13）。

```
shell@Nexus5 /: # mount | grep cgroup
/acct cgroup rw,relatime,cpuacct 0 0
none /sys/fs/cgroup tmpfs rw,seclabel,relatime,mode=750,gid=1000 0 0
none /dev/cpuctl cgroup rw,relatime,cpu 0 0
```

输出结果 2-13 Nexus 5 中与 cgroups 相关的 mount

Bionic 在每个进程启动时，通过/acct 为其设置了 CPU 使用记时器（因此它能应用到系统中所有的进程上）。/sys/fs/cgroup/memory 可以被 ActivityManager（通过 android.os.Process 及其 JNI 方法 setSwappiness）访问。最后，是/dev/cpuctl 目录，尽管它位于/dev 目录下，但仍是个由 Android 调度策略（scheduling policy）安装的 cgroup 目录。在/init 启动时，它会安装该目录并创建其下的各个子目录——/dev/cpuctl/tasks 对应系统任务，/dev/cpuctl/apps/tasks 对应运行在前台的应用，/dev/cpuctl/apps/bg\_non\_interactive/tasks 对应运行在后台的应用，并以此对各个组进行调度。每个组都被分配到一个“共享”CPU 的数值，并被赋予了一个可以运行的时间上限。这样就能防止因有意或无意的错误操作，导致某个进程完全占据整个 CPU 运行时间的情况发生。/dev/cpuctl 的配置是在/init.rc 中完成的，相关代码如代码清单 2-2 所示。

代码清单 2-2 /init.rc 脚本中安装 cpuctl cgroups 的部分

```
mkdir /dev/cpuctl
mount cgroup none /dev/cpuctl cpu
chown system system /dev/cpuctl
chown system system /dev/cpuctl/tasks
chmod 0660 /dev/cpuctl/tasks
write /dev/cpuctl/cpu.shares 1024
write /dev/cpuctl/cpu.rt_runtime_us 950000
write /dev/cpuctl/cpu.rt_period_us 1000000
mkdir /dev/cpuctl/apps
chown system system /dev/cpuctl/apps/tasks
chmod 0666 /dev/cpuctl/apps/tasks
write /dev/cpuctl/apps/cpu.shares 1024
write /dev/cpuctl/apps/cpu.rt_runtime_us 800000
write /dev/cpuctl/apps/cpu.rt_period_us 1000000
mkdir /dev/cpuctl/apps/bg_non_interactive
chown system system /dev/cpuctl/apps/bg_non_interactive/tasks
chmod 0666 /dev/cpuctl/apps/bg_non_interactive/tasks
write /dev/cpuctl/apps/bg_non_interactive/cpu.shares 52
write /dev/cpuctl/apps/bg_non_interactive/cpu.rt_runtime_us 700000
write /dev/cpuctl/apps/bg_non_interactive/cpu.rt_period_us 1000000
```

## debugfs

debugfs 文件系统是用于（输出）内核级的调试信息的。驱动以及类似的子系统可以自由地把驱动的调试信息转储到这个文件系统中。和其他伪文件系统一样，如果文件系统已经被 mount



了，大量的调试信息就能像读取其他文件那样被读取出来。

不过请注意，debugfs 没有必要一定要被 mount 到系统中，而且内核也可以被编译成不支持 debugfs 的形式。如果内核支持 debugfs，它就可以用下面这行简单的命令行命令（通常是在 /init.hardware.rc 中执行）mount 上来：

```
mount -t debugfs none /sys/kernel/debug
```

尽管理论上可以把它 mount 到任意一个 mount 点上，但是因为它实在是太有用了，所以通常都能在 “/” 目录中找到它的符号链接。比如，在模拟器镜像中，就有一个 “/d” 符号链接指向 debugfs 的 mount 点。

debugfs 中的内容是完全由内核的版本以及内核中实现了哪些 debug 特性决定的。表 2-20 中给出的是在各个版本的 Android 内核中常见的几个文件/目录。

表 2-20 /sys/kernel/debug 目录中的文件/目录

文件/目录	用 途
binder	提供通过 Android IPC（进程间通信）机制中的 binder 方式传递的大量数据 <sup>1</sup>
tracing	好用到难以置信，提供由 Linux 内核的 ftrace 机制产生的海量的调试和跟踪信息
wakeup-sources	内核级的定时器，用在 Android 系统或驱动程序防止设备休眠时

functionfs(/dev/usb-ffs/adb)

在 Android 系统中，USB 的功能经常会需要根据用户的选择（即，是以 USB 调试、大容量存储介质还是以其他方式连接设备，用户的这一选择将通过 init 传递进来，详见第 4 章）动态地进行重新配置，它是由一个特定的 “gadget” 驱动进行控制的。

传统的驱动程序（Android L 之前的内核版本）需要通过 sysfs 导出它的重新配置参数。这一做法是它被认为非常臃肿，并需要改进的原因之一。

functionfs 的引入：在 2010 年某个时候，一个相对较新的特性被引入 Linux 内核——由 Linux 内核提供一个通用的文件系统，使驱动能够获取用户态空间中发出的对配置修改的请求。这一文件系统可以被认为是 对 sysfs 的一个补充，只不过设计后者的目的是向用户态输出一些内核中的变量和驱动信息，而前者的设计目的是从用户态获取输入。root 用户可以使用 mkdir(2) 创建目录，这会引起内核创建相应的内核对象，这些对象可以在稍后由在用户态中发起的、对目录中的伪文件进行的 write(2) 操作予以初始化。

1 Binder 是 Android 系统进程间通信（IPC）方式之一。——译者注

## procfs(/proc)

procfs 文件系统名副其实——它提供了一个基于目录的观察系统中运行的进程的方式。这个想法最初是出现在 Plan 9 操作系统上的，随后 Linux 马上吸收并改造了它，使之能提供大量关于进程、线程以及其他全方位的系统诊断信息。事实上，在一些有争议的观点中，/proc 已经成了一个诊断文件堆放场了——因为最初 Linux 在这个目录中只提供了一些伪文件接口。

不论在 /proc 中提供太多东西到底是不是件好事，这都不妨碍它成为一个极为重要的文件系统。许多 Linux 实用程序（如 top、netstat、lsof 和 ifconfig）以及许多 Android 工具（如 procrank、librank）都把它作为诊断信息的来源，没有它就不能运行。Linux 在 proc(5) 的手册页中记录了相当详细的相关信息，并保持它不断更新。我们将在第 7 章中讨论 procfs 在调试中的用法。

## pstore(/sys/fs/pstore)

pstore 机制是 Linux 的一个内核特性（在 3.5 版时引入），它允许内核把部分物理内存（RAM）单独划为 persistent store 区。它对于一种特定的应用——抓取内核崩溃（panic）时的数据，非常有用。

内核崩溃是指内核中出现了内存破坏的情况。由于这类情况发生之后可能会影响到文件系统的执行逻辑。所以，这时任何向文件系统写入数据的操作，都可能使情况进一步恶化，甚至导致文件系统也被破坏掉。UNIX 系统通常会把内核崩溃时的数据转储到 swap 分区里去，但这也不能保证重启之后数据一定都还在。而且 Android 是没有 swap 分区的，因此，剩下的唯一靠谱的解决方案是：专门为此留出一部分物理内存（即一个专用的内存区域，persistent store），让内核把它崩溃时的数据（至少是 bare minimum）转储到这里来。随后内核自动执行一次热重启（也就是中间不切断电源的重启），这也就意味着物理内存中的信息不会在重启的过程中而丢失。在重启的过程中，内核会去检查 persistent store 区，看看里面是不是留有上一次系统运行遗留下来的数据。如果有的话，它就会通过 /sys/fs/pstore，让我们能在用户态中读取到这些数据。

在较旧版本的 Android 系统中，这一功能是利用了一个被称为“RAM console”的 Android 特有的特性（也算是某种 Android 内核技巧）提供的。追溯它的实现代码，我们发现它还是位于 /init.rc 中。这段代码会从 /proc/apanic\_console 和 /proc/apanic\_threads 中获取数据，并把它们写到 /data/dontpanic 中（这一大圈绕的，简直就是“银河系漫游指南”）。随着 pstore 功能的到来，这一实现方式已经被放弃，转而使用 /sys/fs/pstore。

### 实验：检查 Android L 中的 pstore

在 Android L 系统中（或者其他内核版本号为 3.10 或者大于 3.10 的系统中），pstore 很可能

是默认开启的。要检查它是不是开启了，可以去检查 `/sys/fs/pstore`，或者其他被用于 pstore 文件系统的 mount 点是否存在（见输出结果 2-14）。

```
root@flounder:/ # mount | grep pstore
pstore /sys/fs/pstore pstore rw,relatime 0 0
root@flounder:/sys/fs/pstore # ls -l
-r--r----- system log          107902 2014-12-25 14:46 console-ramoops
```

输出结果 2-14 pstore 的 mount 位置


如果你的内核最近重启或崩溃过，该 mount 点中会有一个文件：`console-ramoops`，其中存有最近一次的 `dmesg` 的输出。该文件的拥有者所属组为 `system:log`（这是由 `/init.rc` 脚本设置）。这一权限设置使得该文件的内容可以在 `adb shell`（它是 `log` 组的成员）中被读取出来。连上 `adb shell` 之后，一直到重启，你都可以用“`cat /sys/fs/pstore/console-ramoops`”命令获取内核 ring buffer 最近一次输出的数据。

不过，如果你冷启动了你的系统，这个目录可能就是空的了。在这种情况下，你可以用 `adb reboot` 命令重启系统，或者（如果你有胆子玩的话）也可以用下面这条命令，强行让内核崩溃一回：

```
echo c > /proc/sysrq-trigger
```

这样，那个文件就又会出现了。

---

 `/proc/sysrq-trigger` 伪文件是个极其有用（但也非常危险）的 `/proc` 文件。该文件是只能写的，用 `echo` 向它写入一个字符，就等价于同时按下很少有人知道的 `SysRQ` 键+`ATL`+用 `echo` 写入的这个字符对应的键。这是个魔幻般的组合键，不过只在命令行界面下有效。`SysRQ` 功能仅供在系统失去响应时，作为应急通道使用，因为 `sysrq` 请求是由运行在最高优先级上的键盘中断处理器（`keyboard interrupt handler`）处理的。在和这个文件打交道时，请千万多加小心，因为其中大多数功能是只供紧急情况下使用的，可能会有一定的危险性。

---

## selinuxfs(/sys/fs/selinux)

和 `debugfs` 一样，`SELinuxFS` 传统上也是 mount 在 `/sys` 下，但却不是 `sysfs` 文件系统的一部分。这个文件系统是专供 `SELinux` 使用的，其中存储了与安装策略（`installed policy`）相关的文件。

我们将在第 8 章中详细讨论 `SELinux`，这里仅仅先蜻蜓点水式地介绍一下，这个文件系统中最重要的文件是 `policy` 和伪文件 `enable`。其中，`policy` 提供了加载（编译，可以使用的二进制可执行文件的格式）安全策略，`enable` 则用来切换这些策略是否要被强制执行（事实上 `getenforce/setenforce` 工具集中的工具也是通过它来实现相关功能的）（见输出结果 2-15）。



```
root@flounder:/ # getenforce
Permissive
root@flounder:/ # echo 1 > /sys/fs/selinux/enforce
root@flounder:/ # getenforce
Enforcing
root@flounder:/ # setenforce 0
root@flounder:/ # cat /sys/fs/selinux/enforce
0
```

输出结果 2-15 演示强制执行一个 SELinux 策略

sysfs(/sys)

如果以字母顺序排序，sysfs 可能只能排在最后了，但从重要性上说，它的重要性仅次于 procfs。sysfs 是在 Linux 内核版本 2.6 中作为对 procfs 的补充而被引入的，为了能把 /proc 里的东西整理得井井有条些，把与硬件和模块相关的配置文件移到一个单独的目录中去，并让目录的层次也更清晰些。

你会发现，/sys 是个“整洁的”目录，伪文件被分门别类地放在各自所属的子目录中的地方。你能看到的这些子目录已经列在表 2-21 中了。

表 2-21 /sys 中的子目录（门类）

子目录	内 容
block	存放块 I/O 层（Block I/O Layer）的控制文件。每个块设备都有自己的子目录，其中记录了其所属的 I/O 调度程序等参数
bus	设备，按连接它的总线类型分类存放，每种总线类型（比如 i2c/、mmc/、soc/）一个子目录
class	设备，按设备类型分类存放，每种设备类型（比如 input/、sound/）一个子目录
dev	设备，按设备读写方式分类存放，每种读写方式（比如 block/、char/）一个子目录
devices	设备，按设备在设备树（device-tree）上的分类类型存放
firmware	供有固件升级功能的设备使用
fs	供文件系统驱动使用。这里的有些子目录就是 mount 点（比如 pstore/、selinux/就是刚才讨论的一些伪文件系统的 mount 点），其他一些子目录中（比如 ext4/）则提供文件系统导出的参数和使用情况统计信息
kernel	由各个（内核）子系统提供的一些内核参数，其中的 debug/是 debugfs 的 mount 点
module	每个模块一个子目录，含有模块的统计信息以及（从用户态能够查看，有时甚至是可以修改的）模块参数（如果有的话）
power	电源管理（power management）的统计信息和设置，Android 的 WakeLocks 就是在这里（通过 wake_lock 和 wake_unlock）实现的

不同的设备之间硬件配置的差异非常大，所以各种不同设备中的 `sysfs` 里存放的文件间的差别也非常大。Android 框架之所以能够免于受到因不同厂商选用不同的五花八门的硬件而带来的麻烦，完全要感谢硬件抽象层 (HAL, Hardware Abstraction Layer, 它是由 `/system/lib/libhardware.so` 及其插件构成的)，因为硬件抽象层把对不同特定文件的调用封装成了更为统一的 API (对 HAL 的更进一步的讨论详见第 2 本)。

其他一些设备的标准化程度要更高一些，比如，位于 `/sys/devices/system/cpu/cpu#/cpufreq` 目录中的 CPU 主频调节器 (用来调整主频) 数据，以及供振荡器 (有些设备上会使用振荡器) 使用的 `/sys/class/timed_output/vibrator` 目录。这里可以做一个简单而有趣的实验：你可以试试向这个目录中的 `enable` 这个 `sysfs` 伪文件写入一个比较大的值 (比如 5000)，看看系统会有什么反应。

## 本章小结

在本章中，我们逐个讨论了 Android 中使用的分区和文件系统。而且，我们着重讨论了那些在各种 Android 设备上一般都能找到的分区，有一些甚至是实际上不能被 mount 的分区。接着我们讨论了两种主要的文件系统——`/system` 和 `/data`，我们详细讨论了其中存储的内容以及子目录，甚至是子目录中的子目录的用途。最后，本章还提及了 Linux 伪文件系统，其中存储了海量的诊断和配置文件。这些文件，特别是它们在调试中的用途，还将在本书中 (特别是第 7 章中) 被提及。

下一章的内容 (讨论 Android 的启动和 recovery 过程) 是建立在本章基础上的。那些不能被 mount 的分区，特别是 `aboot` 和 `boot` 分区，由于它们在设备启动过程中扮演重要的角色，将会 (在下一章中) 被仔细地研究。另一个可以 mount 但却很少使用的分区 `/cache`，由于它在 OTA 升级过程中所起的主要作用，也会被详细地讨论。

## 参考文献

- [1] XDA Developers on f2fs: <http://forum.xda-developers.com/showthread.php?t=2697069>
- [2] Samsung f2fs presentation: [http://elinux.org/images/1/12/Elc2013\\_Hwang.pdf](http://elinux.org/images/1/12/Elc2013_Hwang.pdf)
- [3] Linux Weekly News on f2fs: <http://lwn.net/Articles/518988/>
- [4] XDA Developers "El Grande Partition Table Reference": <http://forum.xdadevelopers.com/showthread.php?t=1959445>
- [5] Companion Article: HTC WeakSauce Exploit: <http://NewAndroidBook.com/Articles/HTC.html>
- [6] Android Developer on the jobb utility: <http://developer.android.com/tools/help/jobb.html>
- [7] Android Developer on APK Expansion

Files:<http://developer.android.com/google/play/expansion-files.html>

[8] Android Explorations on JB App Encryption: <http://nelenkov.blogspot.com/2012/07/usingapp-encryption-in-jelly-bean.html>

[9] Linux kernel documentation on CGroups:  
<https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt> (or the kernel sources)



# Android 的启动、备份和重置

和对电脑启动过程的理解一样，大多数的用户想当然地认为他们手机的启动过程也只不过是按在电源键上几个几秒钟，然后桌面或者锁屏界面就会出现在他们眼前。只有在手机升级（“刷机”）或者没电了时，这一启动的过程才会变得有所不同。而在极少数情况下，手机甚至会无法启动——也就是大家常说的“手机变砖头”了。

尽管总体来说，手机的启动过程和电脑基本上是一致的，但是相对而言，手机的启动过程更加复杂，也会涉及更多环节。再加上使用 Android 系统厂商有那么多，他们各自又生产了大量不同型号的手机，这就使得精细地描述这一过程会非常冗长，而且它在不同型号/生产厂商制造的设备中还都不一样。所以本章中描述这一过程时，把重点放在这些各种不同设备共同拥有的那些特性上。

我们首先会去分析 Android 系统镜像。镜像中的各个组件会被分别刷到手机中对应的分区中（详见上一章中的讨论）。你可以手动下载这些系统镜像，另外手机也可以通过 OTA（Over-The-Air）升级的方式获取到它。尽管厂商可以自行决定用什么格式，以及怎样生成这些镜像，但是大多数系统镜像还是遵循着一个通用结构——由一个 Boot Loader，一个（含有内核和 RAM disk 的）bootimg 镜像以及多个 `simg` 格式的分区镜像组成。我们也将按照这些组件在系统启动或 `recovery` 过程中所扮演的角色，依次讨论它们。

在讨论过启动操作的流程之后，再去讨论一下它的逆过程——关机，也是很有必要的。尽管在 Linux 层上，关机和重启操作都非常简单（它们都是由一个系统调用来处理的），但是为了能够控制屏幕（比如弹出电源按键菜单），并重启到 `recovery` 模式，Android 中的操作要更加复杂些。

讨论了如何启动到 `recovery` 模式下之后，很自然，我们就会想要知道 `recovery` 是如何执行的，以及升级操作的步骤是怎样的——所以在下一节中就将去讨论 OTA 升级包和 OTA 升级过

程。

最后，我们将讨论定制固件镜像的方法——我们常常误称其为“ROM”<sup>1</sup>。我们将讨论如何修改或者整个地替换掉那些关键的组件。在这一讨论过程中，我们有意省略掉了一个自然的推论——root 手机的技术。我们将在第 8 章（在这章中，我们将从安全角度分析 Android）里讨论它。

本章的学习将要用到一个名为 imgtool 的工具。这是一个用来查看和提取 android 系统镜像中各个组件的工具。为了方便你下载和使用，我把这个工具的源码和预先编译好的可执行文件一起打包进了一个压缩包，放在本书的官网上供你下载。

## 3.1 Android 系统镜像

各种 Android 设备都只能刷专门为相应型号的设备定制的镜像。厂商会提供一套系统镜像，把它作为“出厂默认”的 Android 系统刷在设备上。一个完整的系统镜像是由以下几个文件组成的，刷机时它们会被写入各自对应的分区中。

- **Boot Loader:** 其中含有在启动阶段由应用处理器（application processor）<sup>2</sup>执行的代码。这些代码一般是用来寻找和加载 boot 镜像的，但同时也被用来进行固件升级和让系统启动到 recovery 模式下。大多数 Boot Loader 还会实现一个小小的 USB 栈（USB stack），通过它，用户在电脑上就可以控制启动和升级过程（通常是通过 fastboot）。Boot Loader 会被刷到“aboot”分区里去，不过在有些手机上（比如 HTC）这个分区会被称为“hboot”分区。
- **boot 镜像:** 它一般是由内核和 RAM disk 组成的，作用是加载系统。假设启动正常，RAM disk 会被 Android 用作 root 文件系统(/)，其中的/init.rc 及相关文件规定了系统余下的其他部分该如何被加载。boot 镜像会被刷到“boot”分区里去。
- **recovery 镜像:** 这个镜像同样是由内核和（另一个）RAM disk 组成的，一般用来在正常

---

1 从技术上说，ROM 指的是只读存储介质（Read-Only-Memory），存储在这类介质上的数据是不可更改的，或者（如果是 EEPROM 的话）只有在特殊情况下才能被擦除和重写。Android 手机或其他 Android 设备中确实有一个真正的 boot ROM 硬件，用于存放启动最开始阶段中要执行的代码（根据下文内容加上这一句），不过启动过程的其余部分则是由存储在闪存芯片上各个分区中的程序/数据完成的——只要你有足够的权限，就能轻而易举地向这些分区写入数据。——原注

2 Android 系统中处理器应该被理解为一个芯片组，除了这个应用处理器之外，还有显卡、GSM 等一系列处理器，其中这个应用处理器就相当于 PC 中的 CPU 这个角色。——译者注

启动失败或者通过 OTA 升级时，把系统加载到“recovery 模式”。这个镜像会被刷到“recovery”分区上。

- **/system 分区镜像：**这里存放的是一个完整的 Android 系统，其中除了谷歌提供的二进制可执行文件和框架（framework）之外，还有厂商和/或运营商提供的类似的东西。
- **/data 分区镜像：**这里存放的是“默认出厂设置”的数据文件，它们是/system 分区中程序正常运行所必需的文件。当手机需要重置到“出厂状态”时，也只需把这个分区镜像刷写到/data 分区，覆盖掉原来的数据即可。

谷歌公司把供它自己的 Nexus 系列的手机、平板电脑使用的各种系统镜像放在了“预装系统镜像仓库”（factory image repository）<sup>[2]</sup>中。我希望你能在阅读本章时，使用这些镜像或者你自己手机/平板电脑的镜像，跟着我完成各个实验，获取第一手的感性经验。如果你的设备已经 root 了，你可以用第 2 章中介绍的方法，从一个正在运行的手机/平板电脑中安全地提取出各个分区的二进制镜像。或者你也可以按照下列步骤，解压谷歌公司提供的镜像。

1. 从谷歌的“预装系统镜像仓库”网站上下载镜像。下载下来的应该是文件名类似下面这个样子的一个用 gzip 压缩了的 tar 文件：

piscine\_devicename-build-factory-first\_32\_bits\_of\_SHA1\_checksum.tgz

2. 用 tar 命令解压这个文件，命令的执行结果应该类似于输出结果 3-1。

```
morpheus@Forge (~/.Images) $ tar xzvf hammerhead-ktu84p-factory-35ea0277.tgz
x hammerhead-ktu84p/
x hammerhead-ktu84p/image-hammerhead-ktu84p.zip
x hammerhead-ktu84p/radio-hammerhead-m8974a-2.0.50.1.16.img # Baseband update (best left alone)
x hammerhead-ktu84p/bootloader-hammerhead-hhz11k.img # Bootloader components
x hammerhead-ktu84p/flash-all.bat # Batch file (Windows)
x hammerhead-ktu84p/flash-all.sh # Flash all images
x hammerhead-ktu84p/flash-base.sh # Bootloader and radio only
#
# Proceed to unpack the main image
#
morpheus@Forge (~/.Images) $ cd hammerhead-ktu84p
morpheus@Forge (~/.../ktu84p) $ unzip image-hammerhead-ktu84p.zip
Archive: image-hammerhead-ktu84p.zip
  inflating: boot.img
  inflating: recovery.img
  inflating: system.img
  inflating: userdata.img
  inflating: cache.img
  inflating: android-info.txt
```

输出结果 3-1 解压 Nexus 5 的预装系统镜像文件

接下来我们将依次讨论系统镜像的各个组成部分（radio/基带除外）。



## Boot Loader

尽管 Android 设备的厂商可以随心所欲地去实现一个自己的启动加载器 (Boot Loader)，但它们大多数还是选用了“LK”(Little Kernel) 启动加载器 (三星是一个值得注意的例外)。LK 启动加载器并不是 Android 源码树的一部分，但它还是可以从 CodeAurora<sup>[3a]</sup>和 googlesource<sup>[3b]</sup>那里 (至少部分地) 下载到。

顾名思义，LK 只实现了启动功能中最基本的那一部分。“LK”这个名字稍稍有点误导人，因为它并不是一个 Linux 内核，而只是一个可启动的运行在 ARM 处理器上的二进制镜像。LK 中只实现了启动加载器所应实现的最小的那部分功能，其中包括：

- **基本的硬件支持**——这是在 LK 的 dev/ (屏幕帧缓存，按键和使设备可以被用作 USB 目标设备的驱动等基本通用驱动)、platform/ (SoC/芯片组驱动) 和 target/ (设备中特定硬件的驱动) 等源码子树中提供的，没有这些，其他的一切全都免谈。
- **找到并启动内核**——编写所有启动加载器的目的都是找到 bootimage (下文会详细讨论)，并把它的组成部分，也就是内核的镜像文件、ramdisk 和设备树 (device tree) 解析出来，然后用给定的命令行，把系统的控制权交给内核。这一任务是由 app/about 来完成的。
- **基本的 UI**——如果用户中断了本该自动完成的启动过程 (一般是通过执行 adb reboot boot loader 命令，或者在设备刚开始启动时马上按下特定的组合键来中断启动过程的)，about 会提供一个很简单的文字界面，用户可以用设备上的物理键完成操作 (用音量增大/减小键上下切换当前选中项，用电源键确认执行选中项对应的操作)。但是这会儿触摸屏还不能用了。
- **支持 console**——尽管在上市销售的手机/平板电脑中很少可以使用 console 接口连接电脑<sup>1</sup>，但是开发母板还是支持通过串口 (RS232/UART) 提供 console 功能的。LK 在 lib/console (它会被 app/shell 调用) 中提供了一个命令解释器 (运行在一个单独的线程中)，并且支持用户往里面添加新的命令。而在 lib/gfxconsole 中还提供了字体 (font) 之类的基本图形功能。
- **使设备可被用作 USB 目标设备**——这使得 Boot Loader 可以通过一个名为 fastboot 的简洁的协议 (本章将会在稍后讨论它) 与电脑进行通信。我们可以在 app/about/fastboot.c 中找到这个协议的一个基本实现框架，不过厂商可以根据自己的需要，在这个协议中加

---

<sup>1</sup> 令人惊奇的是：确实可以通过 console 连上某些手机 (比如谷歌的 Nexi)，而且还是用我们最猜不到的地方连 console 的——耳机插口！甚至还有教你怎样把 Nexus 4<sup>[4a]</sup>和 Nexus 9<sup>[4b]</sup>上的耳机插口变成 RS232 接口的文档<sup>[4]</sup>。——原注

入它们自己的（OEM）扩展命令。

- 支持闪存分区——由于在系统升级和重置手机（recovery）的过程中，需要 Boot Loader 能够擦除或重写某些分区，所以 LK 通过 lib/fs，也支持基本的文件系统。
- 支持数字签名——为了能够加载用 SSL 证书做过数字签名的镜像，LK 在 lib/openssl 源码子树中，集成了 OpenSSL 项目的一部分代码。

## Boot Loader 镜像

Boot Loader 也可以像其他系统镜像一样更新或重刷。虽然官方没有提供描述这一镜像格式的文档，但是 Android 源码树里的 releasetools.py 脚本（该文件所在目录的具体位置与设备型号有关）还是给出了 Boot Loader 镜像的文件头的格式。这就使我们能够解析 Boot Loader 镜像，并提取出它的各个组成部分。比如输出结果 3-2 就是解析谷歌 Nexus 5 的 Boot Loader 得到的结果。

```
morpheus@Forge (~)% imgtool Images/hammerhead-kot49h/bootloader-hammerhead-hhz1lk.img
Boot loader detected
6 images detected, starting at offset 0x200. Size: 2568028 bytes
Image: 0      Size: 310836 bytes      sb11      # Secondary Boot Loader, stage 1
Image: 1      Size: 285848 bytes      tz        # TrustZone image
Image: 2      Size: 156040 bytes      rpm       # Resource Power Mgmt
Image: 3      Size: 261716 bytes      aboot     # Application Boot Loader
Image: 4      Size: 18100 bytes       sdi       #
Image: 5      Size: 1535488 bytes     imgdata   # RLE565 graphics used by boot loader
```

输出结果 3-2 Nexus 5 的 Boot Loader 镜像

就像输出结果中显示的那样，Boot Loader 镜像是由几个会被刷到各自对应的分区里去的子镜像组成的。狭义上的 Boot Loader 就是指“aboot”——它是应用处理器（application processor）的 Boot Loader。但除此之外，在 Boot Loader 镜像中还有资源电源管理（Resource Power Management）的引导模块（rpm）、ARM TrustZone 镜像（tz）和次级 Boot Loader（sb11）（这些都将在本章的稍后部分中予以讨论）。

Boot Loader 中的这些组件使用的是什么文件格式，并没有正式的文档描述。它们是与处理器的体系结构紧密相关的，比如上面这个例子中对应的处理器是高通的“骁龙”处理器（msm 芯片组）。本次讨论的重点是 aboot，把它提取出来之后，Linux 中的 file(1)命令会误把它识别为运行在 Hitachi SH 大端机处理器上的 COFF 可执行文件。而事实上，这只是因为这个文件的开头部分——一个 40 个字节（某些情况下可能会更大些）的私有的头部——的一些数据恰巧与运行在 Hitachi SH 大端机处理器上的 COFF 可执行文件头部中的数据一致罢了。这个头部的格式如表 3-1 所示。

表 3-1 aboot 的私有头

偏移量	字 段	作 用
0x00	文件签名 (Magic)	0x00000005 (常量)
0x04	版本	版本号 (2 或是 3)
0x08	?	填零
0x0c	镜像加载基地址	把镜像的剩余部分加载到这个指定的虚拟内存地址上
0x10	镜像大小	aboot 镜像的大小
0x14	代码大小	aboot 中代码的大小
0x18	最后一条指令的地址	镜像加载基地址+代码大小
0x1C	数字签名的大小	数字签名的大小 (通常是 0x100 = 256 个字节)
0x20	被加载到内存里去的最后一个字节的地址	最后一条指令的地址+数字签名的大小
0x24	证书链	证书链的大小 (如果有的话)

紧挨着这个头部后面的是 ARM 的可启动镜像。它会被加载到头部中指定的内存地址上去。在这个镜像的开始位置上存放的是一些 ARM 异常处理向量，也就是一些跳转指令，当发生某种异常（比如中断、异常或者停机之类的）时，处理器就能根据这些指令自动跳转到相应的代码上去完成异常处理。根据 LK 对入口点的定义，这些指令中的第一条是 reset 的处理函数。下面这个实验中演示了去除 aboot 镜像头部的办法。

### 实验：去掉 aboot 镜像的头部

如果你手头有一个 Nexus 5 的 ROM 升级包，你可以用 imgtool 工具从 bootloader.img 文件中提取出 aboot 的镜像文件<sup>1</sup>（bootloader.img 的提取方法见输出结果 3-2）。或者，如果你有一台已经 root 了的手机，你也可以用第 2 章给出的方法，提取出 aboot 分区的镜像。只不过在提取时，你要用 aboot 所在的分区（比如/dev/block/mmcblk0p6）换掉第 2 章的例子的那个/dev/mmcblk0。此外，你还得把提取出来的 aboot 镜像存放到一个文件中去（见输出结果 3-3）。

<sup>1</sup> 原文如此，但是输出结果 3-2 中给出的只是分析镜像的结果，要想使用 imgtool 提取 Boot Loader 中的各个组件，你还在这条命令后面加上一个“extract”参数。——译者注



```
morpheus@Forge (~/...-kot49h/)% od -A d -t x4 aboot | head -5
      Magic          Version          NULL          ImgBase
0000000 00000005 00000003 00000000 0f900000
      ImgSize        CodeSize        ImgBase+CodeSize  SigSize
0000016 0003fe2c 0003e52c 0f93e52c 00000100
      ImgBase+CodeSize+SigSize  Certs
0000032 0f93e62c 00001800 ea000006 ea00351c
```

输出结果 3-3 用 od 命令解析 aboot 镜像

我们可以很方便地认出一些形为“eaXXXXXX”的 ARM 指令，因为“ea”是 ARM 中的 B（branch，跳转）指令的操作码。在 ARMv7 中，这张异常向量表占 7×4 个字节，所以 reset 异常处理的起始位置一般都位于 ea000006（和上图给出的结果一样吧）之后 6×4 个字节的位置上。

在把文件最开头的 40 个字节去掉之后（用“dd bs=40 skip=1”命令），文件剩下的部分可以直接放到反汇编器里去分析了。此外，头部中“代码大小”字段指定了代码部分的大小，文件后面多出来的部分是数字签名和证书部分（它们的大小也应该和头部中规定的值相吻合），你应该把它们也给去掉（见输出结果 3-4）。

```
morpheus@Forge (~/...-kot49h/)% dd if=aboot of=aboot.sans.header bs=40 skip=1
morpheus@Forge (~/...-kot49h/)% dd if=aboot.sans.header of=certs bs=0x3e62c skip=1
..
6144 bytes transferred in 0.000064 secs (96155984 bytes/sec)# 6144 = 0x1800 - all's well
```

输出结果 3-4 从 Boot Loader 镜像中读取证书

此外，你还要把代码部分的加载基地址调整到 0x0f900000（也就是在 Boot Loader 镜像偏移 12 个字节位置上的那个字段指定的值）这个内存地址上。逆向分析 Boot Loader 已经超出了本书的讨论范围，不过我还是在本书的官网上给出了一篇相关的文章<sup>[5]</sup>。

## Boot Loader 的加锁与解锁

Android 设备上的 Boot Loader 一般都会被加锁。也就是说，如果数字签名验证没有通过，手机/平板电脑会拒绝刷机或用更新的镜像启动。厂商会在 ROM 中提供他们的公钥，该密钥被用来建立一条贯穿启动过程始终的信任链。这样，所有的启动组件（从 rpm 到 sbl 再到 Android Boot Loader）就都可以被验证（是否来自厂商或有无遭到修改）了。对这些组件的逆向分析表明，其中都有一个 X.509v3 的证书，以及验证密钥时所必需的 OpenSSL 相关代码。

请不要把 Boot Loader 加锁和 SIM 卡加锁混为一谈，SIM 卡加锁通常是用来确保手机只能用在某个电信供应商的网络中的。有些国家的法律已经规定不许把手机和电信供应商捆绑在一起，但没有法律规定市场上销售的手机中的 Boot Loader 不能加锁。

所以，用户可能可以解锁手机，也有可能不能解锁手机，这都是由手机的生产厂商决定的。有些厂商不让用户解锁，而有些厂商则不然。比如，谷歌的 Nexus 5 和英伟达的 NVIDIA Shield 系列，只要敲入一条“fastboot oem unlock”命令就能解锁，如输出结果 3-5 所示。

```
morpheus@Forge (~)$ fastboot devices
05141138021471071D9E    fastboot
morpheus@Forge (~)$ fastboot oem unlock
(bootloader) Showing Options on Display.
(bootloader) Use device keys for selection.
(bootloader) erasing userdata...
(bootloader) erasing userdata done
(bootloader) erasing cache...
(bootloader) erasing cache done
(bootloader) unlocking...
(bootloader) Bootloader is unlocked now.
OKAY [ 21.337s]
finished. total time: 21.337s
```

输出结果 3-5 解锁英伟达的 NVIDIA Shield 设备的 Boot Loader

另一些厂商走的则是“中间路线”，让手机发出一个挑战（challenge），也就是给出一个密码学令牌（cryptographic token），根据是不是得到了某个特定的响应结果，来决定 Boot Loader 是不是能被解锁。还有些厂商会同时销售加了锁的和没有加锁的手机（三星就是个很有名的例子）。从 Android L（至少是从 Nexus 9）开始，Android 默认的“设置”app，就能让用户通过点击“设置”→“开发者选项”来确定手机/平板电脑是不是要加/解锁。这一功能的实现方式是：手机会根据用户的选择，设置 Boot Loader 能够读取的一个分区中的某个 bit 位为“真”还是“假”，然后 Boot Loader 就能根据这个 bit 位知道自己有没有加锁了。

被解锁之后（如果可能的话），Boot Loader 就能完全控制/data 分区中的信息。所以解锁 Boot Loader 会从根本上危及手机的安全性，因为攻击者拿到手机之后，就能给手机刷一个能绕过用户的开机密码或者锁屏图案的恶意更新包，或者把/data 分区中的数据复制出来，把里面的所有个人信息全部偷走。

如果 Boot Loader 不能解锁，那么至少从理论上说，只要不被 root，手机就是安全的。不过在实践中，Android 也并不是不存在能够 root 它的漏洞。事实上，任何一个 3.13 及以后版本的 Linux 内核漏洞利用代码都有可能被用来 root Android 系统。著名的“TowelRoot”（这个漏洞利用代码是由 GeoHot 公开的）会影响到目前市场上销售的所有 Android 设备，而这还只是好几个一般被称为“一键 root”（相当于 iOS 中的越狱程序）的漏洞利用程序中的一个。我们将在第 21<sup>1</sup>章中讨论这些常被用来 root 设备的漏洞利用程序。


## Boot 镜像

Android 的 Boot 镜像中存储的是操作系统的核心组件——内核和 RAM disk。此外，Boot

---

1 作者针对 Android 系统目前有出三本书的考虑，因此此处的第 21 章是指未来第三本书中预估的章节。  
——译者注

镜像（它是由 Android 源码树中的 `mkbootimg` 创建的）的组成部分还有一个很小的头部、内核命令行、一个 Hash<sup>1</sup> 以及一个可选的（在实践中并不使用的）二级启动加载器。所有这些组件都是闪存页边界（通常为 2KB）对齐的。Boot 镜像可以用它的文件签名“ANDROID!”予以识别，这和上文中我们可以用 Boot Loader 的文件签名“BOOTLDR!”识别 Boot Loader 的原理是一样的。

 严格地讲，厂商并不一定要在它们生产的手机上使用本书中所述的 Boot 镜像的格式存储内核和 RAM disk，所以某些手机/平板电脑或许还使用了不同的方式存储这些信息。比如，HTC 的手机就会在 Boot 镜像中使用它自己的头部（见输出结果 3-6）。出现这一情况的原因，或许是 HTC 手机使用的是它自己的 Boot Loader——HBOOT 吧。尽管如此，你在大多数情况下还是都能用下面给出的这个文件签名“ANDROID!”识别 Boot 镜像的。

```
morpheus@Forge (~) $ od -A x -t c mmcblk0p43 # Dump hex offset + ASCII
0000000 250 2 032 244 ? 213 \0 u O W 220 e I 300 J 235
..
0000100 A N D R O I D      DF ** \0 \0 200 \0 \0
```

输出结果 3-6 HTC boot 镜像的头部

然后你可以使用 `dd` 命令，去掉这个 HTC 自己定制的头部（就像上面的例子里那样——`dd bs=0x100 skip=1`）。

Boot 镜像的格式在 `bootimg.h` 文件中有明确的定义，如代码清单 3-1 所示。

代码清单 3-1 boot\_img\_hdr

```
struct boot_img_hdr {
    unsigned char magic[BOOT_MAGIC_SIZE];
    unsigned kernel_size; /* size in bytes */
    unsigned kernel_addr; /* physical load addr */
    unsigned ramdisk_size; /* size in bytes */
    unsigned ramdisk_addr; /* physical load addr */
    unsigned second_size; /* size in bytes */
    unsigned second_addr; /* physical load addr */
    unsigned tags_addr; /* physical addr for kernel tags */
    unsigned page_size; /* flash page size we assume */
    unsigned unused[2]; /* future expansion: should be 0 */
    unsigned char name[BOOT_NAME_SIZE]; /* asciiz product name */
    unsigned char cmdline[BOOT_ARGS_SIZE];
    unsigned id[8]; /* timestamp / checksum / sha1 / etc */
};

/*
** +-----+
** | boot header | 1 page
** +-----+
** | kernel | n pages
** +-----+
*/
```

1 在使用本书官网提供的 `imgtool` 工具解析 `boot.img` 或 `recovery.img` 时，此 Hash 显示为 ID。——译者注



```

** | ramdisk          | m pages
** +-----+
** | second stage    | o pages
** +-----+
**
** n = (kernel_size + page_size - 1) / page_size
** m = (ramdisk_size + page_size - 1) / page_size
** o = (second_size + page_size - 1) / page_size
**
** 0. all entities are page_size aligned in flash
** 1. kernel and ramdisk are required (size != 0)
** 2. second is optional (second_size == 0 -> no second)
** 3. load each element (kernel, ramdisk, second) at
**    the specified physical address (kernel_addr, etc)
** 4. prepare tags at tag_addr. kernel_args[] is
**    appended to the kernel commandline in the tags.
** 5. r0 = 0, r1 = MACHINE_TYPE, r2 = tags_addr
** 6. if second_size != 0: jump to second_addr
**    else: jump to kernel_addr
**
*/

```

## 内核

与大多数操作系统的内核不同，Linux 的内核大多是经过压缩的。内核镜像文件格式（也被称为 zImage 格式）规定，其中必须含有用来把内核镜像中其他经过压缩的部分解压到内存中去的自解压代码。由于不同压缩算法各有所长，所以可选的压缩算法也有好几个，具体使用哪种算法是在 build 内核的过程时（make config）决定的，如表 3-2 所示。

表 3-2 内核文件格式

压缩算法的签名	对应的压缩算法	说 明
\x1f\x8b\x08\x00\x00\x00\x00\x00	GZip	最常用的压缩算法
\x89LZO\x00\x0d\x0a\x1a\x0a	LZO	比 GZip 更快，但是压缩的效率要低 10%~15%。三星使用的是这种算法

内核总是会先运行自解压代码，这也就意味着，我们得先搜索一下文件，找到压缩算法的签名。传统上，大多数 ARM 内核使用的都是 zImage，尽管严格意义上说并不一定要这样做。本书官网上提供的 imgtool 这个程序可以（在需要时）自动解压使用 GZip 或 LZO 算法压缩的内核镜像，提取出能够直接拿去反汇编或者搜索其中字符串的二进制文件。此外，如果你要把它放到一个反汇编器里分析的话，还请记得把镜像的加载基地址改成 0xC0000000（我假设你分析的是一个 32 位的内核）。

内核是 Android 系统中与体系结构最为紧密相关的部分。尽管其他的一些组件也需要关心处理器的类型（系统中使用的到底是 ARM、Intel 还是 MIPS 处理器），但是内核还要关心主板类型和芯片组的型号。因为事实上，移动设备的处理器是个单晶片系统（SoC, System-on-Chip，它由一组芯片构成），其中还含有其他芯片，内核也必须为这些芯片提供专用的驱动。这些驱动

是源码树的一部分，谷歌为不同的芯片组提供了不同的内核设备树（device tree），如表 3-3 所示。

表 3-3 谷歌设备的芯片组设备和主板名称（board name）

项目名称	芯片组厂商	设备（主板名称）
goldfish (M:Ranchu)	N/A	Android emulator
msm	Qualcomm MSM	Nexus One, Nexus 4, Nexus 5 (hammerhead)
omap	TI OMAP	Pandaboard, Galaxy Nexus, Glass (notle)
samsung	Samsung Hummingbird	Nexus S
tegra	NVIDIA Tegra	Motorola Xoom, Nexus 7&9, NVIDIA Shield
exynos	Samsung Exynos	Nexus 10 (manta)

谷歌自己设备的名字就是其主板的项目名，它们的内核镜像可以用 git，从 <https://android.googlesource.com/device> 上的子树中获取，内核的源码（它当然肯定是开源的）也可以用下面这条命令，以类似的方式获取：

```
git clone https://android.googlesource.com/kernel/platform_project.git
```

更详细的描述，请参见 Andriod 的官网文档<sup>[6]</sup>。另外，除了查表 3-3，查看内核中的字符串和符号也是一个确定某个设备的内核是源自何方的好办法。

## ARM 中的设备树（Device Tree）

大多数 ARM 内核需要靠设备树（device tree）文件向内核提供硬件设备定义的相关信息。这个文件提供了设备相互连接的层次关系，使得内核能够据此启动相应的设备。设备树文件一般会被附加在内核镜像的尾部，但有时也会为它专门分配一个分区。

设备树是一个二进制的 blob 文件<sup>1</sup>，可以通过它的文件签名 0xd00dfeed 予以识别。完整地讨论设备树已经超出了本书的论述范围（因为它是一个 ARM 特性，而非 Android 特性），ePAPR specification 和 Thomas Pettazoni 写的一个 PPT 是关于这种文件格式的非常好的文档。你也可以使用 imgtool 这个工具从你的内核镜像中提取设备树文件。具体操作步骤如下面这个实验所述。

### 实验：从 boot.img 中提取设备树文件

imgtool 这个工具除了能解压 boot.img，把其中的内核和 ramdisk 提取出来之外，也会自动

---

1 如果你是跟着本书分析 Nexus 5 的系统镜像的话，那么它就是下面这个实验中提取到的 devicetree.dtb 文件。——译者注

地把内核镜像中的设备树文件提取出来（如果能找到它的话）。不过提取出来的是个纯二进制的文件（.dtb 文件，可以用它的文件签名 0xd00dfeed 识别它）。为了反编译得到明文文本形式的设备树，你还需要使用 dtc 工具（它是 Ubuntu 系统中 device-tree-decompiler 包或 Fedora 系统中 dtc 包的一部分）（见输出结果 3-7）。安装完这个工具之后，只需一条命令就能反编译这个文件，得到文本格式的 .dts 文件。

```
morpheus@Forge (~/.Android/Book) % imgtool Images/hammerhead-kot49h/recovery.img extract
Part      Size      Pages      Addr
Kernel:   8331496
Ramdisk:  1095649
Secondary: 0
Tags:     2700000
Flash Page Size: 2048 bytes
Name:
CmdLine:  console=ttyHSL0,115200,n8 androidboot.hardware=hammerhead user_debug=31 maxcpus=2
Extracting contents to directory: extracted/
Looking for device tree... Found DT Magic @6bb8e8 - will extract tree(s)
Found GZ Magic at offset 18612 - will extract kernel
morpheus@Forge (~/.Android/Book) % cd extracted
#
# Get the device tree compiler package. This example is on a Fedora system, so use yum
# On Ubuntu, you'd likely use "sudo apt-get install device-tree-compiler"
#
morpheus@Forge (~../extracted) % yum whatprovides dtc
Loaded plugins: langpacks, refresh-packagekit
dtc-1.4.0-2.fc20.x86_64 : Device Tree Compiler
Repo                : fedora
morpheus@Forge (~../extracted) % sudo yum install dtc
--> Running transaction check
--> Package dtc.x86_64 0:1.4.0-2.fc20 will be installed
...
Installed:
    dtc.x86_64 0:1.4.0-2.fc20
Complete!
morpheus@Forge (~../extracted) % dtc -I dtb devicetree.dtb -O dts -o devicetree.dts
morpheus@Forge (~../extracted) % more devicetree.dts
/dts-v1/;
/ {
    #address-cells = <0x1>;
    #size-cells = <0x1>;
    model = "LGE MSM 8974 HAMMERHEAD";
    compatible = "qcom,msm8974";
    interrupt-parent = <0x1>;
    qcom,msm-id = <0x7e 0x96 0x20002 0xb>;
    chosen {
    };
    aliases {
        spi0 = "/soc/spi@f9923000";
        spi7 = "/soc/spi@f9966000"; # ...
```

输出结果 3-7 从 Nexus 5 的 boot.img 中提取和反编译设备树文件

## RAM disk

Boot 或 recovery 镜像的另一个组件就是“initial RAM disk”，它通常也被简称为 initrd。RAM disk 提供了最初的文件系统，在操作系统启动时，它被用作根文件系统（/）。它会和内核一起



被 Boot Loader 预加载到 RAM（也就是内存，所以得名 RAM disk）中去，所以是可以直接访问而不需要经过任何特殊的驱动的。这并不是一个 Linux 特性，我们知道，在其他一些 UNIX 系统中也使用它，其中最著名的当属 iOS 了，尽管 iOS 中是把它和 kernelcache 并排放在 .ipsw 系统镜像里的。

传统上，initramfs 是用来提供内核操作时所需的相关设备的驱动程序的。在它的帮助下，Linux 发行版本才能提供一个通用的、相对紧凑的内核，并在最开始安装的过程中，把所需的驱动（这些驱动因各种不同的硬件配置而差异非常大）打包放进一个单独的文件中。为了解开“先有鸡还是先有蛋”（加载驱动先要能访问存储器，而要能访问存储器又先要有存储器的驱动）这个死结，一些关键的必备驱动会被打包放在 initramfs 中。这样，内核就能直接在 RAM 中访问到它们了。initramfs 中还含有一个启动程序（/init），内核会把它作为系统中的第一个进程（PID=1）运行起来，用于完成一些需要在用户模式（user mode）下执行的初始启动操作（比如加载模块）。

当 RAM disk 的操作完成之后，Linux 通常都会丢弃掉它，以便使用磁盘上的文件系统（用一个通常被称为“pivoting root”的进程）。但在 Android 中，initramfs 却还是会被保留在内存中，用来提供根文件系统（/）。由于其中的文件会被经常访问，而且 RAM disk 本身所占的内存空间也非常的小，所以这样做还是很有用的。此外，这么做也使得修改根文件系统的难度变得更大了一些，因为 Boot 镜像是有数字签名保护的。

Linux 支持两种文件格式的 RAM disk——initrd（ext4 文件系统镜像）和 initramfs（CPIO 文件）。后者更为常用，不过常常也会被称为 initrd。CPIO 文档是一种只需占用很少内存空间的简单格式。为了进一步节省空间，该文档还会经过 gzip 压缩（内核是支持 zlib 的，因为内核要用它把自己解压出来）。

尽管理论上厂商可以随心所欲地 build 自己的 RAM disk，但是大多数厂商还是选择修改 Android 模拟器的镜像。所以当你在某些设备上看到 init.goldfish.rc 时，也请不要大惊小怪……因此大多数 RAM disk 彼此也是非常相似的。更进一步说，在同一个手机/平板电脑中，Boot 镜像和 recovery 镜像中的 RAM disk 绝大部分内容也是一样的，只有在控制系统启动过程的/init.rc 文件中会有一些微妙的差别。在 recovery 镜像的 RAM disk 中，/init.rc 中删掉了一些标准的服务，同时又留下了 adbd 和/sbin/recovery。

根据相关规定，内核在被压缩之后会和 RAM disk 一起放在一个单独的分区中。这背后有一个非常重要的设计思想：把两个东西打包在一起，只要用一个数字签名，就能同时保护这两个东西。即，只要应用一次防篡改措施，就能保护两者都不会被篡改。内核当然是系统的关键组件，但 RAM disk 也是非常重要的，其中的/init 及其对应的/init...rc 文件控制着系统的启动过

程。/init 会以 root 权限启动，负责启动所有其他的系统组件，只要修改一下/init.rc 文件就能获取设备的 root 访问权限，但只要搞不定数字签名，这一切就是水中花，并中月。

### 实验：解压 RAM disk

使用 imgtool 这个工具，你可以从 Boot 镜像或者 recovery 镜像中提取出 RAM disk。然后，就像这个实验中演示的那样，只要使用标准实用程序——gunzip 或者 cpio，就能把它解压出来（见输出结果 3-8）。如果你手头没有 Boot 镜像，你可以试试从 Android 模拟器镜像中把它提取出来。

```
morpheus@Forge (~/.Android/Book) % imgtool Images/hammerhead-kot49h/recovery.img extract
Part      Size      Pages    Addr
Kernel:   8331496
Ramdisk:  1095649
Secondary: 0
Tags:     2700000
Flash Page Size: 2048 bytes
CmdLine: console=ttyHSL0,115200,n8 androidboot.hardware=hammerhead user_debug=31 maxcpus=
Extracting contents to directory: extracted/
#
# File is easily recognizable as a gzip archive
morpheus@Forge (~/.Android/Book) % file extracted/ramdisk
extracted/ramdisk: gzip compressed data, from Unix
#
# Create a temporary directory to extract archive to
morpheus@Forge (~/.Android/Book) % mkdir tmp;cd tmp
# If your system doesn't have gunzip, you can use gunzip first
# cpio switches: -i(input) -d(directory) -v(verbose)
morpheus@Forge (~/.Android/Book/tmp) % gunzip ../extracted/ramdisk | cpio -ivd
charger
..
init.rc
sbin/adbd
sbin/healthd # Starting with KitKat, healthd is also in RAM disk
sbin/recovery # Note recovery binary - instrumental for restore/update
```

输出结果 3-8 用 imgtool 从 recovery 镜像中提取 RAM disk，并解压之

至于 initramfs（即根文件系统）中都存了些什么东西，详见上一章中的表 2-6。在下面的实验中，我们还会去比较 Boot 镜像中的 RAM disk 和 recovery 镜像中的 RAM disk。

## /System 和/Data 分区镜像

/system 和/data 分区我们在上一章里已经讨论过了。因为厂商可以提供自己专门开发的分区刷写工具，所以，用哪种镜像文件格式存储它们完全可以由厂商自己说了算。不过由于大多数厂商使用的还都是 fastboot，所以他们很可能也就沿用了谷歌在自己的镜像上使用的 `simg` (sparse image) 格式。能处理这种文件格式的实用程序可以在 AOSP 的 `system/core/libsparse` 目录下找到。

在 `simg` 文件格式中，文件的开头部分是一个很小的头部（只占 28 个字节）。该头部中记录了镜像文件的相关元数据，其格式如表 3-4 所示。

表 3-4 `simg` 镜像的头部

偏移量	长 度	字段含义
0	4	文件签名
4	4	版本号（实际上是 <code>major_version+minor_version</code> <sup>1</sup> ），当前的值总是 0x00000001
8	2	头部的大小，总是 28
10	2	数据块的大小
12	4	簇 <sup>2</sup> 的大小，对于 Ext 文件系统来说，这个值一般是 0x1000（4KB）
16	4	文件系统中簇的个数
20	4	镜像文件中数据块的个数
24	4	校验和（可选，通常全是零）

### 实验：把 Android 的/system 镜像 mount 到电脑上

使用本书官网上提供的 `imgtool` 工具，可以很方便地把 `/system` 镜像提取出来。你也可以用源码把 AOSP 的 `simg2img` 工具编译出来。这里演示时使用的是 `system.img`，因为 `userdata` 和 `cache` 镜像基本上是空的（见输出结果 3-9）。

Android 模拟器镜像（位于 `$SDK_ROOT/system-images` 中）就是纯二进制的文件系统 `dd` 镜像<sup>3</sup>，你可以直接用 `loop` 参数把它 `mount` 上来。在这一章后面还有个实验，在那个实验中我们会演示把这个实验反过来做——修改 `/system` 分区镜像，以便把这个改过的镜像再刷回到手机中。

---

1 根据 `sparse_format.h` 文件中的定义，这 4 个字节实际上是分为两个部分，前 2 个字节是 `major_version`，后两个字节是 `minor_version`。——译者注

2 `block` 和 `chunk` 都是数据块，一个是指镜像文件中的数据块，一个是指文件系统中的数据块，因为在 FAT 文件系统中这个数据块有个读者更熟知的术语“簇”，所以将文件系统中的数据块译为“簇”，以示区别。——译者注

3 这个术语，指用 `dd` 或类似工具，以位对位方法复制出来的镜像。——译者注



```

Forge (~/.ssh/hammerhead-lpv79)$ file system.img
system.img: data  # unrecognized :- (
Forge (~/.ssh/hammerhead-lpv79)$ od -A x -t x4 system.img | head -2
00000000 magic: ed26ff3a version: 00000001 chnk/hdr: 000c001c blksize: 00001000
00000010 blocks: 00040000 chunks: 00000595 checksum: 00000000 0000cac1
# Using the simg_dump Python script you can get some details about the sparse image:
#
Forge (~/.ssh/hammerhead-lpv79)$ $SDK_ROOT/system/core/libsparse/simg_dump.py system.img
system.img: Total of 262144 4096-byte output blocks in 1429 input chunks.
# Compile simg2img. Yes, you can do that through the NDK build system. But..
# this works without having to configure anything (or even download the NDK)
Forge (~/.ssh/hammerhead-lpv79)$ cd $SDK_ROOT/system/core/libsparse
Forge (~/system/core/libsparse)$ gcc backed_block.c output_file.c sparse*.c simg2img.c \
> -I./include -lz -o simg2img
sparse_read.c:122:10: warning: implicit declaration of .. #... whatever..
# We now have simg2img! Go back and unpack image:
#
Forge (~/system/core/libsparse)$ cd -
Forge (~/.ssh/hammerhead-lpv79)$ $SDK_ROOT/system/core/libsparse/simg2img system.img system.ex
# simg2img isn't the talkative type - but will generate the output file
#
Forge (~/.ssh/hammerhead-lpv79)$ file system.ext4
system.ext4: Linux rev 1.0 ext4 filesystem data (extents) (large files)  # Success!
# Compare the sparse image to the full, raw image: Note decent savings
#
Forge (~/.ssh/hammerhead-lpv79)$ ls -lh system.img system.ext4
-rw-r--r-- 1 morpheus staff 1.0G Jun 27 07:09 system.ext4
-rw-r--r-- 1 morpheus staff 668M Jan 1 2009 system.img
# Now we can mount filesystem as a loop device (need to be root for this step)
#
Forge (~/.ssh/hammerhead-lpv79)$ sudo mount -o loop system.ext4 /mnt
Forge (~/.ssh/hammerhead-lpv79)$ ls /mnt
app  build.prop  fonts  lib  media  recovery-from-boot.p  vendor
bin  etc  framework  lost+found  priv-app  usr  xbin

```

输出结果 3-9 解压 Android 系统镜像中的/system 分区镜像

## 3.2 启动过程

介绍完系统镜像的所有组件之后，现在我们可以把注意力放在实际启动的过程上来了。尽管不同手机/平板电脑的启动过程不会完全相同。但是整个启动的过程还是可以大致划分成如图 3-1 所示的几个阶段。

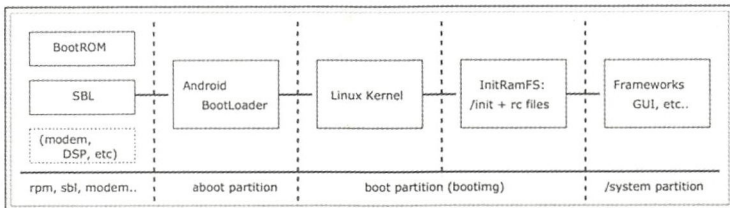


图 3-1 Android 的通用启动流程

### 固件启动过程

手机中的固件，就相当于 PC 机里的 BIOS（或者现在的 EFI），其主要部件是一个由硬件制

造厂商提供的 boot ROM。顾名思义，boot ROM 是记录在只读（read-only-memory）部件中的，所以它一般都很小，只记录了一小段在手机刚一加电到加载 sbi 之前必须要先执行的启动代码。它的作用是初始化其他的硬件部件，使它们处于至少是可以使用的状态。然后，boot ROM 就会加载 sbi（secondary boot loader，次级启动加载器）并把系统控制权交给 sbi。sbi 是个软件，所以它不受硬件 ROM 容量的限制，可以记录更多的数据，所以也能去执行更复杂的初始化任务（比如，显示一个启动画面）。

手机里的处理器和 PC 机里的还不一样，它并不是一个单独的 CPU（比如说一块 Intel 或者 ARM 处理器），而是一个完整的芯片组（SoC，System-on-Chip）。事实上，这也就意味着有好几个处理器在同时工作，而应用处理器（application processor）只是其中之一。比如说，高通的“骁龙”处理器中就至少含有 4 个子处理器：RPM（资源/电源管理）、Krait（应用处理器）、Adreno（图形处理器——GPU<sup>1</sup>）和 Hexagon（数字签名处理器——DSP，Digital Signal Processor）。所以 MSM 芯片组的启动也是一个特别冗长的过程——先执行 boot ROM 中的 PBL（初级启动加载器，primary boot loader），然后再加载执行 sbi（次级启动加载器）。而这个 sbi 本身又可以分为三个阶段（sbi1→sbi2→sbi3），各个 sbi 阶段加载和验证下一阶段代码的过程又是一段令人眼花缭乱的复杂操作<sup>2</sup>，其中还涉及了 rpm 和 tz（ARM 可信区域）中的代码。之后应用处理器才能启动其他的组件，执行 aboot 中的代码——一直要到到这儿，Android 的 Boot Loader 才算是正式登场。

## fastboot 协议

Android 系统的 Boot Loader 大多都支持“fastboot”协议，谷歌把它作为 Android 的一部分，放在官网上供下载<sup>3</sup>。fastboot 是一个简单的、基于文本的协议，这意味着它能使用在手机/平板电脑与电脑连接的 USB 信道上。从性能角度讲，这个协议并不快（比如它是同步的协议），协议名中的“fast”应该是指它实现起来很容易（所以实现起来也很快）。图 3-2 中给出了在电脑和手机之间传递协议消息的过程。

---

1 原文为 CPU，显然是 GPU 之误。——译者注

2 说它是“令人眼花缭乱的”是因为这确实是个非常复杂的操作。除了高通内部的保密文档之外，根本没有完整的文档说明，高通内部的文档也只流出来了一部分。你懂的，这事不能说太细。不过从上述文档中所能得到的信息以及对它的详细讨论，可以在 XDA-Developers 论坛上的 2 个帖子（见本章参考文献[9a]和[9b]）里找到，我只能帮你这么多了……——原注

3 厂商没有必要一定要支持 fastboot，它们也可以使用自己的 Boot Loader 协议以代替 fastboot，或者作为 fastboot 的补充。三星的 ODIN 和亚马逊的 Boot Loader 就是这样的例子。——原注

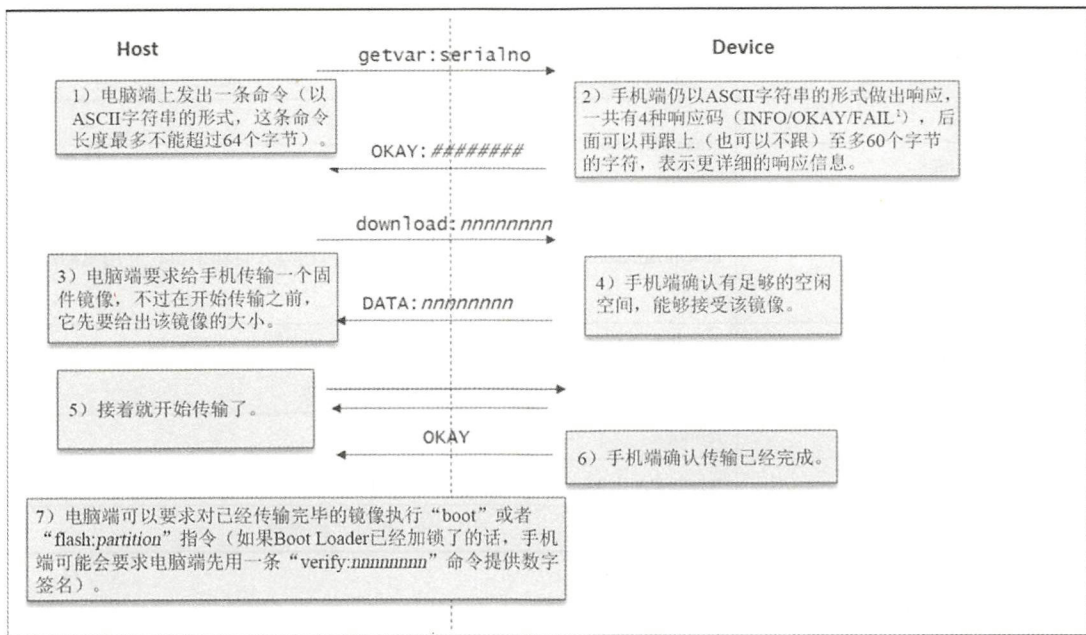


图 3-2 fastboot 的实现

本书编写时 fastboot 的协议版本号是 (0.4)，在 system/core/fastboot/fastboot\_protocol.txt 中给出了这个协议的实现细节，表 3-5 中列出了电脑端二进制可执行文件所使用的命令行参数格式及其对应的协议命令。

表 3-5 fastboot 的默认命令

命令行命令	对应的协议指令	描 述
flash <partition> [<filename>]	download: %08x <sup>2</sup> , flash:partition	用指定的分区镜像文件刷指定的分区
flash:raw boot <kernel> [<ramdisk>]		用内核创建一个 booting，并把它刷到手机中，并启动之
flashall		一次性刷 boot、recovery 和 system 分区
update		用升级包重刷手机/平板电脑

- 1 原文如此，按照 system/core/fastboot/fastboot\_protocol.txt 中的讲法，最后一种响应码应该是 DATA。——译者注
- 2 按照 system/core/fastboot/fastboot\_protocol.txt 中的讲法，这个“download: %08x”的作用是把相关数据写入内存，供接下来的 flash、boot、ramdisk 指令使用。fastboot.exe 列出的帮助信息中的“download”都是指这个 download 指令，而不是“去网上下载”的意思。——译者注



续表

命令行命令	对应的协议指令	描 述
erase <partition>	erase:partition	擦除闪存上的指定分区
format <partition>		格式化闪存上的指定分区
getvar <variable>	getvar:variable	显示 Boot Loader 中由 variable 参数指定的变量的值
boot <kernel> [<ramdisk>]	download:%08x, boot	把参数中指定的内核写入目标手机，并用它启动目标手机
devices	getvar:serialno	列出所有已连上电脑的手机/平板电脑
continue	continue	让目标手机继续启动
reboot	reboot	正常重启手机
reboot-bootloader	reboot-bootloader	重启手机，并停在 Boot Loader 上
oem [command [args]]	command[:args]	发送一条 OEM 扩展命令

### 实验：使用 fastboot

Android SDK 中提供的这些命令组成了一个简单但却完备的协议。为了确认你的设备的 Boot Loader 是不是支持 fastboot，你首先需要让设备在启动到 Boot Loader 阶段时停下来。要让设备在这时停下，除了在启动时按下规定的组合键（具体是哪个组合键，会因设备的不同而略有差异）之外，你也可以使用“adb reboot bootloader”命令。这样你的设备就会重启到 Boot Loader，然后（如果它支持 fastboot 的话，在你输入“fastboot devices”命令时，就会看到设备的序列号输出在屏幕上）这个输出结果（见输出结果 3-10）和你平时输入“adb devices”命令时看到的结果应该是一样的。

```
morpheus@Forge (~)% adb reboot bootloader
morpheus@Forge (~)% $SDK_ROOT/platform-tools/fastboot devices
FA43BSF00073    fastboot
```

输出结果 3-10 “fastboot devices” 命令的输出结果

这时，手机/平板电脑的屏幕上显示的应该是 Boot Loader 的 UI，你只需使用物理按键就能对 Boot Loader 菜单进行操作（通常用音量增加/降低键上下切换当前选中项，电源键选择执行当前选中项）。你也可以用表 3-5 中给出的任何一条命令操作你的手机，不过如果你不理解所输入命令的含义的话，乱输命令可能会破坏手机/平板电脑上的数据。比如，你可以输入“getvar all”命令，列出 Boot Loader 中的所有变量。在不同的手机上，这条命令的输出结果将会是不一样的，输出结果 3-11 给出的是在一台 HTC One M8 手机上执行该命令的输出结果。

```

morpheus@Forge (-)% $SDK_ROOT/platform-tools/fastboot getvar all
(bootloader) version: 0.5
(bootloader) version-bootloader: 3.16.0.0000
(bootloader) version-baseband: 0.89.20.0222
(bootloader) version-cpld: None
(bootloader) version-microp: None
(bootloader) version-main: 1.12.605.11
(bootloader) version-misc: PVT SHIP S-ON # HTC: S(ecurity)-[ON/OFF]
(bootloader) serialno: Phone Serial #
(bootloader) imei: Device ID
(bootloader) imei2: Not Support
(bootloader) meid: Device ID (same as IMEI)
(bootloader) product: m8 wlv
(bootloader) platform: hTCBmsm8974      # Note msm chipset
(bootloader) modelid: 0P6B20000
(bootloader) cidnum: VZW_001
(bootloader) battery-status: good
(bootloader) battery-voltage: 0mV
(bootloader) partition-layout: Generic
(bootloader) security: on
(bootloader) build-mode: SHIP
(bootloader) boot-mode: FASTBOOT
(bootloader) commitno-bootloader: 3a4162f9
(bootloader) hbootpreupdate: 11
(bootloader) gencheckpt: 0
all: Done!

```

输出结果 3-11 一台 HTC-One M8 手机上执行命令的输出结果

不过, fastboot 真正令人感兴趣的是它的 oem 扩展部分。你可以输入“fastboot oem”命令, 从而得到一张所有可用命令的列表(显然在不同的手机上, 这张表的内容是不一样的)。这些命令是非常多样和有用的——HTC 支持 dmesg (获取 Boot Loader 的日志)、get\_temp (读温度传感器)、read/writeusername (定制手机名称) 和 read/writeimei (供运营商配置手机联入他们的网络) 等命令。上文中已经提过, 有些手机——特别是 Nexus 5 和英伟达的 Shield 系列, 支持“oem unlock”命令。这条命令让你能解锁手机的 Boot Loader, 让你的手机能自由地刷各种定制的固件镜像。

## 内核启动过程

Android 内核的启动过程和 Linux 的没什么太大区别。不过, 坏消息是, 后者经常会发生一些变动。比如, 内核升级时也会经常添加或删除一些组件, 而且不同平台之间变化也很大。因此这一节中只对 3.x 内核的启动过程做一个大致的描述, 你或许还需要去参考某个特定设备的内核源码树中的代码。

再说一遍, Boot Loader 的作用是将经过压缩的内核 zImage 和 RAM disk 加载到内存中去。一旦加载完毕, 系统的控制权就会被转移到 zImage 的入口点上去。在这个时刻, 内核还处于压缩状态, 所以实现在 arch/architecture/boot/compressed/head.S 文件(如果是一台 x86 或 x64 设备, 对应的文件则是 head\_32.S 或 head\_64.S)中, 入口点上的指令就是去调用 decompress\_kernel 函

数，这个函数会在屏幕上显示出一条我们熟悉的“Uncompressing Linux... done, booting the kernel”消息，并在解压完内核之后，把控制权交给“真正的”入口点函数。再说一遍：这是个与处理器体系结构紧密相关的函数，是用汇编的方式写就的（写在 arch/arm/kernel/head.S 或者 arch/x86/kernel/x86/kernel/head\_[32|64].S 文件里的）。接下来，还要在底层执行设置 MMU（内存管理单元）和页表（方便切换到虚地址上去）等操作，最后才能把系统的控制权交给内核的 main 函数——start\_kernel()。

start\_kernel()这个函数与处理器的类型无关，所以它被放在了 init/main.c 中实现。从某种意义上说，这个函数写得相当好，其中几乎没有使用任何变量，大多数的启动工作都是通过调用其他函数的方式完成的。为了尽可能地长话短说，start\_kernel()先用各个专门的函数去初始化所有重要的框架（framework），然后再去调用 rest\_init()函数——顾名思义，这个函数的作用就是去初始化剩下的一切东西。这个函数会生成一个 kernel\_init 线程，该线程负责初始化各个不同的子线程。

因为有这么多的子系统要初始化，要是没什么控制机制，内核代码也就肯定会变得极其冗长。initcall 机制提供了一个非常漂亮的解决方案：它定义了表 3-6 中的 8 个初始化级别，使得 kernel\_init 线程可以（通过 do\_basic\_setup()中的 do\_initcalls()函数）依次调用它们。

表 3-6 initcall 定义的层级

#	级 别	注 释
0	early	用于生成初始化辅助线程，比如 RCU、SoftIRQs 和工作队列
1	core	供 binfmt 和 sockets 之类的“核心”（core）子系统使用
2	postcore	供 bdi（块设备刷写线程，block device flush threads）和 kobjects 使用
3	arch	这一级别上都是与处理器体系结构相关的初始化工作
4	subsys	供 bio、crypto 和 sound 这些普通的子系统使用
5	fs	供给用来支持文件系统的 VFS 层使用
6	device	供驱动和普通模块使用，module_init 宏也是映射在这一级别上的
7	late	分配给应该在启动的最后一个阶段中初始化的东西（高级内存管理、oops 处理函数等）使用

这一机制的设计思想与经典的用户模式下的 init 进程的“运行级”（runlevel，init 进程用它把各个子系统的启动脚本划分在不同的组中）的概念是非常相似的。initcalls 模仿这一思想的方式是：要求各个子系统在注册它们的初始化函数时，通过 level\_initcall 宏<sup>1</sup>，说明自己应该被放入哪个级别。在系统启动过程中，kernel\_init 就会依次初始化各个级别中注册的系统。执行到哪

1 斜体的“level”应该被替换为要注册的层级。——译者注



个级别，就会调用注册在该级别上的各个子系统的初始化函数，当所有级别都被处理完之后，整个内核的初始化也就完成了。

内核启动过程中输出的消息可以用 `dmesg(1)` 命令查看。不过，由于内核是在一个环形缓冲区 (ring buffer) 中记录这些信息的，所以当你执行获取 root shell (在手机中执行 `dmesg` 命令必须要有 root 权限) 的操作时，这些信息有可能会被覆盖掉一部分 (该环形缓冲区的大小是在 build 内核时指定的)。

与其费心劳力地一步一步跟踪整个执行流程，用下面这个代码清单 3-2 说明问题就比较轻松直观了。这个代码清单中给出的是各个子系统初始化函数向 `dmesg` 中输出的内容，从中我们可以看到，各个子系统确实是被依次初始化的。加粗的行是与处理器体系结构无关的，所以不论是在 ARM 移动设备还是在 x86 移动设备上，你应该都能看到它们 (尽管具体的值可能会稍有不同)。

代码清单 3-2 一台 Android 模拟器上的 `dmesg` 命令的输出结果 (已加注释)

```
<6>Booting Linux on physical CPU 0 # smp_setup_processor_id
<6>Initializing cgroup subsys cpu # cgroup init early
<5>Linux version 3.4.0-gd853d22 (nnk@nnk.mtv.corp.google.com) ... # pr notice("ts",
<4>CPU: ARMv7 Processor [410fc080] revision 0 (ARMv7), cr=10c53c7d
<4>CPU: PIPT / VIPT nonaliasing data cache, VIPT nonaliasing instruction cache
<4>Machine: Goldfish
<5>Truncating RAM at 00000000-7fffffff to -2f7fffff (vmalloc region overlap).
<4>Memory policy: ECC disabled, Data cache writeback
<7>On node 0 totalpages: 194560
<7>free_area_init_node: node 0, pgdat c04a7394, node_mem_map c084f000
<7> Normal zone: 1520 pages used for memmap
<7> Normal zone: 0 pages reserved
<7> Normal zone: 193040 pages, LIFO batch:31
<7>pcpu-alloc: s0 r0 d32768 u32768 alloc=1*32768
<7>pcpu-alloc: [0] 0
<4>Built 1 zonelists in Zone order, mobility grouping on. ... # build all zonelists
<5>Kernel command line: qemu.gles=0 qemu=1 console=ttyS0 ... # pr notice("Kernel com
<6>PID hash table entries: 4096 (order: 2, 16384 bytes) # pidhash init()
<6>Dentry cache hash table entries: 131072 (order: 7, 524288 bytes) # vfs caches in
<6>Inode-cache hash table entries: 65536 (order: 6, 262144 bytes)
<6>Memory: 760MB = 760MB total # mem init();
<5>Memory: 764912k/764912k available, 13328k reserved, OK highmem
<5>Virtual kernel memory layout:
<5> ...
<6>NR_IRQS:256 #early_irq init()
<6>sched_clock: 32 bits at 100 Hz, resolution 1000000ns, .. # sched clock init()
<6>Console: colour dummy device 80x30 # console init()
<6>Calibrating delay loop... 412.87 BogoMIPS (lpj=2064384) # calibrate delay()
<6>pid_max: default: 32768 minimum: 301
<6>Security Framework initialized # security init()
<6>SELinux: Initializing. # security initcall(selinux init);
<7>SELinux: Starting in permissive mode
<6>Mount-cache hash table entries: 512 # vfs caches init()
<6>Initializing cgroup subsys debug # cgroup init();
<6>Initializing cgroup subsys cpuctt
```

代码清单 3-2 一台 Android 模拟器上的 dmesg 命令的输出结果（已加注释）（续）

```

<6>Initializing cgroup subsys freezer
<6>CPU: Testing write buffer coherency: ok      # check bugs (macro)
#
# .. from this point on, we're at rest init - and in kernel init thread
# .. this calls do basic setup(), which in turn calls do initcalls()
# .. Note ordering of output is consistent with initcall levels
#
<6>Setting up static identity map for 0x35e610... # early initcall(init static idmap)
<6>NET: Registered protocol family 16      # core initcall(netlink proto init);
<6>bio: create slab at 0                    # subsys initcall(init bio);
<6>Switching to clocksource goldfish_timer
<6>NET: Registered protocol family 2      # fs initcall(inet init);
<6>IP route cache hash table entries: 32768 (order: 5, 131072 bytes)
<6>TCP established hash table entries: 131072 (order: 8, 1048576 bytes)
<6>TCP bind hash table entries: 65536 (order: 6, 262144 bytes)
<6>TCP: Hash tables configured (established 131072 bind 65536)
<6>TCP: reno registered
<6>UDP hash table entries: 512 (order: 1, 8192 bytes)
<6>UDP-Lite hash table entries: 512 (order: 1, 8192 bytes)
<6>NET: Registered protocol family 1
<6>RPC: Registered named UNIX socket transport module. # fs initcall(init sunrpc);
<6>RPC: Registered udp transport module.
<6>RPC: Registered tcp transport module.
<6>RPC: Registered tcp NFSv4.1 backchannel transport module.
<6>Trying to unpack rootfs image as initramfs... # rootfs initcall(populate rootfs)
<6>Freeing initrd memory: 312K
<4>... goldfish new pdev IRQ enumeration...
<4>... goldfish pdev worker interrupt registration..
<6>audit: initializing netlink socket (disabled) # initcall(audit init);
<5>type=2000 audit(0.270:1): initialized
<6>Installing knfsd (copyright (C) 1996 okir@monad.swb.de).
<6>fuse init (API version 7.18) # module init(fuse init);
<7>yaffs: yaffs built Jul 9 2013 17:46:43 Installing.
<6>msgmni has been set to 1494
<7>SELinux: Registering netfilter hooks

```

```

<6>io scheduler noop registered
<6>io scheduler deadline registered      # elv register
<6>io scheduler cfq registered (default)
<4>allocating frame buffer 1080 * 1920, got (null)
<4>goldfish_fb: probe of goldfish_fb.0 failed with error -12
<6>console [ttyS0] enabled
<6>brd: module loaded
<6>loop: module loaded
<6>nbd: registered device at major 43
<4>... goldfish specific stuff...
<6>tun: Universal TUN/TAP device driver, 1.6
<6>tun: (C) 1999-2004 Max Krasnyansky
<4>smc91x.c: v1.1, sep 22 2004 by Nicolas Pitre
<4>eth0: SMC91C11xFD (rev 1) at fe013000 IRQ 13 [nowait]
<4>eth0: Ethernet addr: 52:54:00:12:34:56
<7>eth0: No PHY found
<6>mousedev: PS/2 mouse device common for all mice
<4>*** events probe ***
<4>events_probe() addr=0xfe016000 irq=17
<4>events_probe() keymap=qwerty2
<6>input: qwerty2 as /devices/virtual/input/input0
<6>goldfish_rtc goldfish_rtc: rtc core: registered goldfish_rtc as rtc0
<6>device-mapper: uevent: version 1.0.3
<6>device-mapper: ioctl: 4.22.0-ioctl (2011-10-19) initialised: dm-devel@redhat.com
<6>ashmem: initialized      # ashmem init()
<6>logger: created 256K log 'log_main'      # device initcall(logger init);
<6>logger: created 256K log 'log_events'

```



代码清单 3-2 一台 Android 模拟器上的 dmesg 命令的输出结果（已加注释）（续）

```

<6>logger: created 256K log 'log_radio'
<6>logger: created 256K log 'log_system'

<6>Netfilter messages via NETLINK v0.30.
<6>nf_conntrack version 0.5.0 (11956 buckets, 47824 max)
<6>ctnetlink v0.93: registering with nfnetlink.
<6>NF_TPROXY: Transparent proxy support initialized, version 4.1.0
<6>NF_TPROXY: Copyright (c) 2006-2007 BalaBit IT Ltd.
<6>xt_time: kernel timezone is -0000
<6>ip_tables: (C) 2000-2006 Netfilter Core Team
<6>arp_tables: (C) 2002 David S. Miller
<6>TCP: cubic registered
<6>NET: Registered protocol family 10
<6>ip6_tables: (C) 2000-2006 Netfilter Core Team
<6>IPv6 over IPv4 tunneling driver
<6>NET: Registered protocol family 17
<6>NET: Registered protocol family 15
<6>8021q: 802.1Q VLAN Support v1.8
<6>VFP support v0.3: implementor 41 architecture 3 part 30 variant c rev 0
<6>goldfish_rtc goldfish_rtc: setting system clock to 2014-04-16 01:15:32 UTC (1397610932)
<6>Freeing init memory: 148K
<7>SELinux: 512 avtab hash slots, 1319 rules.
<7>SELinux: 512 avtab hash slots, 1319 rules.
<7>SELinux: 1 users, 2 roles, 288 types, 1 booleans, 1 sensors, 1024 cats
<7>SELinux: 84 classes, 1319 rules
<7>SELinux: Completing initialization.
<7>SELinux: Setting up existing superblocks.
<7>SELinux: initialized (dev sysfs, type sysfs), uses genfs_contexts
... iterates over all mounted filesystems ...
<7>SELinux: initialized (dev sysfs, type sysfs), uses genfs_contexts
<5>type=1403 audit(1397610932.610:2): policy loaded auid=4294967295 ses=4294967295
<4>SELinux: Loaded policy from /sepolicy
<5>type=1404 audit(1397610932.620:3): enforcing=1 old_enforcing=0 auid=4294967295 ses=4294967295
<4>init (1): /proc/1/oom_adj is deprecated, please use /proc/1/oom_score_adj instead.

```

内核启动线程执行完毕之后，就会转入用户态，用 PID 1 启动/init。我们将在下一章里详细讨论/init，以及由它运行的各种 Android 相关的服务。

### 3.3 关机和重启

大多数用户会让手机一直处于开机状态，但是偶尔，用户也会想要重启或者关掉手机。这时，用户一般都会长按电源键，直到屏幕上弹出一个确认关机操作的对话框。如果用户点击关机或者是重启手机（当然还有个选项是使用“飞行模式”），手机就会关机或重启。

不过在这个看似简单的操作背后，具体的实现过程却是相当冗长的，而且涉及相当复杂的动作，如图 3-3 所示（提示：这张图应该从下往上读）。



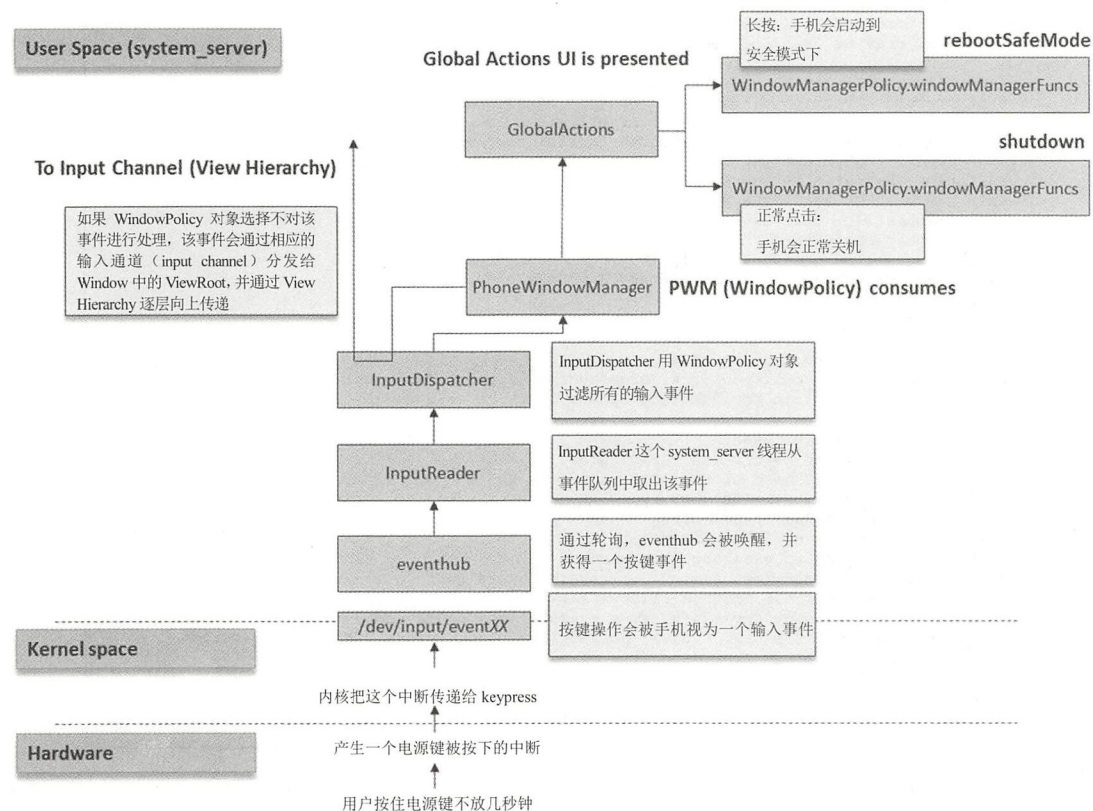


图 3-3 按下电源键后，系统内部的操作流程

按下物理键，会产生一个中断，然后该中断会被 Linux 内核捕获到，并被内核转换成一个输入事件，再作为一个 `EV_KEY/KEY_POWER/DOWN` 事件传给 Android 运行时 (runtime)。和其他所有事件一样，该事件会依次交给 Android 的 `InputReader` 和 `InputDispatcher`（这两个都是 `system_server` 线程），<sup>1</sup> 后者会把该事件传递给 `WindowPolicy` 对象。如果按键的时间足够长（这个时长是定义在 `ViewConfiguration` 的 `GLOBAL_ACTIONS_KEY_TIMEOUT` 上的常量 500 毫秒），默认的 `WindowPolicy` 对象 (`com.android.internal.policy.impl.PhoneWindowManager`) 就会拦截到该事件，并（通过调用 `GlobalActions.showDialog()` 方法）在屏幕上弹出一个菜单。

如果用户确实是要关机，那么还要分两种情况：如果用户只是正常点击一下“关机”虚拟

1 关于 Android 的输入架构，以及底层中断是如何通过 `InputManager`、`WindowManager`、`Policy` 和 `Views` 逐层向上传递的，将在第 2 本中予以详细讨论。——原注

键的话,就会正常关机;但是如果用户长按“关机”虚拟键的话,手机就会重启到安全模式下。这两个操作都是由一个专门的 ShutdownThread 来处理的,其中的 shutdownInner()方法负责在执行具体的关机操作之前弹出一个确认对话框(当然也可以设置成不弹出这个对话框)。

如果用户确认要关机, beginShutdownSequence()要设置两个屏幕唤醒锁(wakelock),以保持屏幕在整个过程中始终处于点亮状态(这是为了提供更好的用户体验)。然后 ShutdownThread 就会运行,它依次要执行的操作如图 3-4 所示。

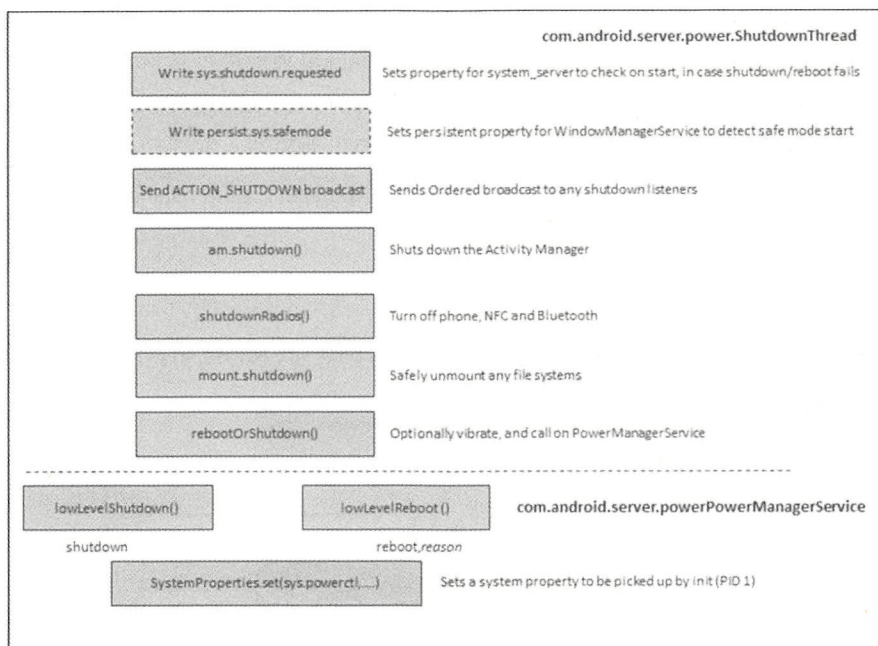


图 3-4 Android 关机时依次执行的操作

关机的最后一步在 Linux 原生层上是由/init 执行的。因为这个线程是负责实现系统属性的(详见第4章),它负责把 sys.powerctl 属性设置为 shutdown 或是 reboot,reason。这里的这个 reason 可以是 recovery, 也可以是 Boot Loader。要是你觉得这几个属性值挺眼熟的话,那一点也不奇怪,因为 adb reboot 也是要求用户在这些值中选择一个,并以用户的输入设置同一个属性(在 adb 中把它称为 ANDROID\_RB\_PROPERTY)的。这样,不管之前选择的是何种操作,终归都会回到/init,而它又会去调用 libcutils 中的 android\_reboot 函数。这个函数实际上只是内核中的 reboot()系统调用(或称 \_\_reboot)的一个封装。\_\_reboot 的调用格式是 Linux 定好了的,上面那个 reason 将被作为参数传递给它。

## 3.4 应用的备份和恢复

就像人吃五谷杂粮免不了生病一样，操作系统也需要处理数据遭到破坏甚至是完全丢失的问题。因此备份和恢复也是操作系统必须提供的一项重要功能；应用（App）也需要这一能力，以保存和还原它们的配置和数据；高级用户也需要这一能力，来备份整个手机，以便在出现问题时，能将手机恢复到系统还原点或一个已知可以正常工作和启动的配置上。

事实上，从 API level 8 开始，Android 就向应用提供了一个名为 BackupManagerService 的框架服务（framework service），它既可以让各个应用各自单独地进行备份，也可以一次性地备份所有应用。该框架服务的内部实现细节以及它所提供的 API（应用程序编程接口）将会在第 2 本中（和其他的框架服务一起）讨论。备份的架构相当地简洁，它让应用自己决定要备份哪些数据。当相关数据发生变化时，应用会通知 BackupManager，然后 BackupManager 会把该应用添加到一个队列中。

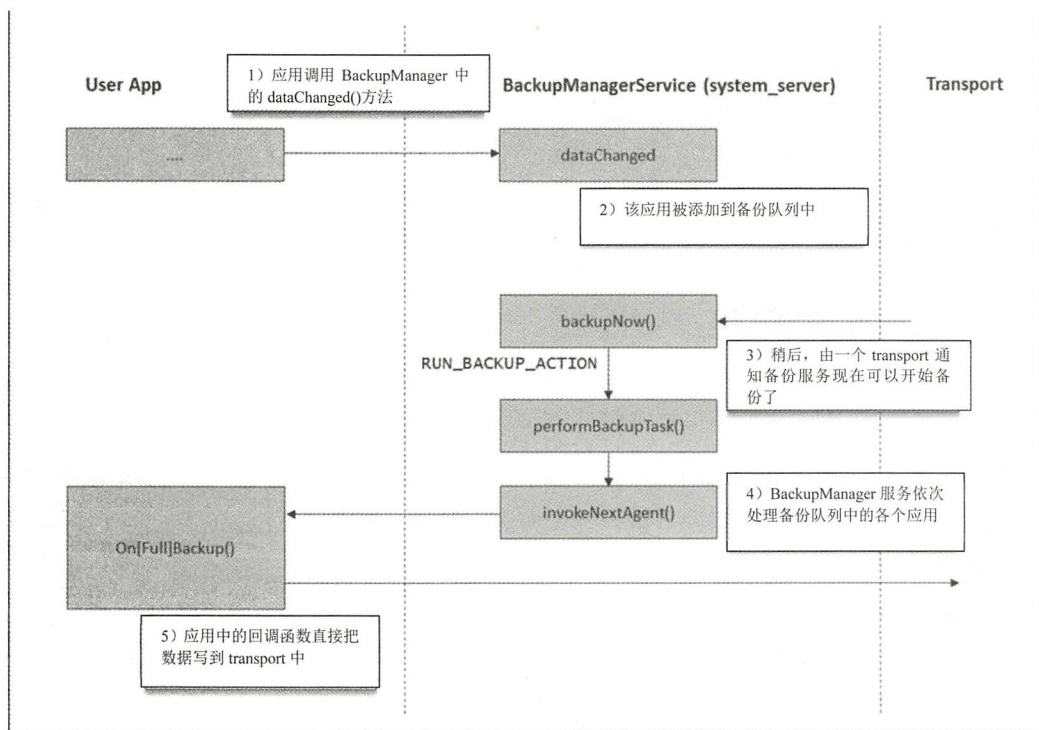


图 3-5 Android 备份架构的简易示意图

稍后，当 BackupManager 得到一个可以进行备份的通知时，它就会创建一个备份集（backup set），把队列中的应用放在一起，对于每个应用，它都要去调用该应用的 onBackup() 回调函数。



BackupManager 会把一个文件描述符通过这个回调函数传递给应用，应用用它写（/读）备份数据。这个 BackupManager 提供的文件描述符实际上是指向一个 transport 的，但这一点对于应用来说是完全透明的。应用根本无须操心通过 transport 读写的数据具体是源自何方的——数据可能是备份在本地的，也有可能被备份到“云端”（比如谷歌云服务器，或者是设备制造商提供的云服务那里）。至于数据到底应该被备份到哪里去这一问题，最终要交给系统（或厂商）来决定。表 3-7 中给出了常用的 transport。

表 3-7 Android 使用的 transport

Transport 名称	备份到
<code>com.google.android.backup/.BackupTransportService</code>	谷歌的云服务器，应用需要提供一个特定的使用该服务的 API key
<code>com.android.server.enterprise/.EdmBackupTransport</code>	企业备份，仅用于被托管的（managed）手机
<code>android/com.android.internal.backup.LocalTransport</code>	手机本地备份

## 命令行工具

从高级用户的角度讲，系统备份和恢复（restore）的接口非常简单，只需借助两个 Dalvik upcall 脚本 `bmgr` 和 `bu`，这两个脚本都需要借助于 Java，通过 Binder 与 BackupManagerService 进行通信（这将在第 7 章中详细讨论）。`bmgr` 有一篇不错的文档<sup>[10]</sup>，而且不加任何参数直接调用 `bmgr`，该脚本会详细地列出它的参数和使用方法。这些参数总结在表 3-8 中。

表 3-8 bmgr 的参数和作用

命令行	作用
<code>backup package</code>	指定下次运行时需要备份的包
<code>enable 0 1</code>	启用/禁用备份服务
<code>enabled</code>	检查备份服务是否已经启用了
<code>list transports</code>	列出所有可用的 transport，已经指定为默认 transport 的 transport 上会标有*（详见表 3-7）
<code>list sets</code>	列出恢复集（restore set）
<code>transport transportName</code>	设置默认的 transport
<code>restore set [App]</code>	从指定的恢复集（restore set）中恢复所有应用或者只恢复指定的应用
<code>run</code>	立即执行之前设定的备份
<code>wipe transportName package</code>	擦掉由 transportName 参数指定的 transport 中 package 参数指定的包（package）中的所有备份数据
<code>fullbackup package</code>	对指定的包（package）做一个完全备份（full backup）

相比之下，`bu` 却是完全没有文档支持的，它输出的结果不是直接给用户的，而是要通过 Android logging 系统才能得到。它只接受一个参数——`backup` 或是 `restore`，不过当这个参数为 `backup` 时，还有几个二级参数（switch）可供指定。可用的二级参数已经列在表 3-9 中了，加粗的部分为默认值。

表 3-9 bu backup 使用的二级参数

二级参数	用 途
<code>-[no]apk</code>	保存或不保存应用的.apk 文件
<code>-[no]obb</code>	保存或不保存应用的.obb(opaque binary blobs)文件
<code>-[no]shared</code>	保存或不保存共享资源
<code>-[no]system</code>	是不是要对系统应用做完全备份（full backup）
<code>-[no]widgets</code>	保存或不保存 widgets
<code>-[no]compress</code>	是否要压缩保存备份
<code>-all</code>	备份所有的东西（需要用户确认）

或许你觉得这几个二级参数有些眼熟，这是因为它们就是传递给 `adb backup` 的参数（尽管后者的文档中并没有提到有 `-nocompress` 这么个参数）。在通过 `adb` 进行备份时，会直接调用 `bu` 这个 upcall 脚本，这也就是为什么它用起来不像 `bmgr` 那样友好的原因。

## 本地备份

输入“`adb backup -all`”命令会触发系统对所有应用做一个完全备份（full backup）。这么做会使得 `bu` 去调用 `BackupManagerService` 的 `fullBackup()` 方法，该方法将弹出一个可以定制的 UI 提示界面给用户（如图 3-6 所示）。

这个默认提示 UI（如图 3-6 所示）的 activity 是写死在 `com.android.backupconfirm` 上的。要在屏幕上看见和操作这个 UI 界面，用户首先必须要点亮屏幕，输入锁屏密码/图案。这样就加了一层安全保险，它让黑客没法在用户不知情的情况下，拿到手机 1、2 分钟，备份出里面的数据，然后再把手机还给用户。此外，这也让用户可以有机会取消备份操作，以及设定备份文件的保护密码。

如果用户同意进行备份操作，屏幕下方就会出现一个 `toast`，提示说备份已经开始了，当前正在进行备份的包的包名也会在 `toast` 中显示出来。

在要把数据备份到和手机相连的电脑上的时候，`adb` 会把移动端上的 `transport` 文件描述符指向电脑上的一个文件，用 `adb backup` 的 `-f` 参数可以指定该文件，否则使用的就是默认的

backup.ab 文件。备份文件使用的是一种私有文件格式，在备份文件加密或是不加密时，其格式还会略有不同。关于这种文件格式的唯一文档是写在 BackupManagerService 类的源码里的，不过其中还是提供了相当详细的信息，如代码清单 3-3 所示。

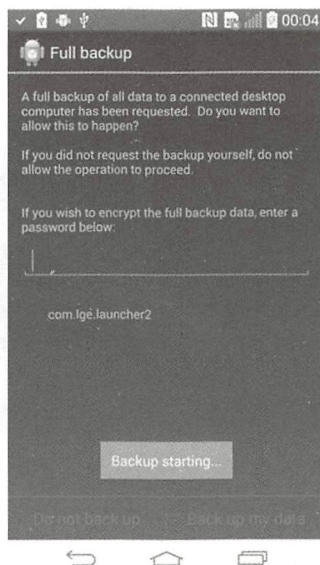


图 3-6 默认的备份 UI (截图源自一台运行 KitKat 的 LG G3 手机)

代码清单 3-3 Android 备份文件的格式

```
// Write the global file header. All strings are UTF-8 encoded; lines end
// with a '\n' byte. Actual backup data begins immediately following the
// final '\n'.
//
// line 1: "ANDROID BACKUP"
// line 2: backup file format version, currently "2"
// line 3: compressed? "0" if not compressed, "1" if compressed.
// line 4: name of encryption algorithm [currently only "none" or "AES-256"]
//
// When line 4 is not "none", then additional header data follows:
//
// line 5: user password salt [hex]
// line 6: master key checksum salt [hex]
// line 7: number of PBKDF2 rounds to use (same for user & master) [decimal]
// line 8: IV of the user key [hex]
// line 9: master key blob [hex]
//      IV of the master key, master key itself, master key checksum hash
//
// The master key checksum is the master key plus its checksum salt, run through
// 10k rounds of PBKDF2. This is used to verify that the user has supplied the
// correct password for decrypting the archive: the master key decrypted from
// the archive using the user-supplied password is also run through PBKDF2 in
// this way, and if the result does not match the checksum as stored in the
// archive, then we know that the user-supplied password does not match the
// archive's.
```



## 实验：查看 Android 备份文件的内部结构

使用代码清单 3-3，可以很轻松地把 Android 备份文件头部中各个字段的含义判读出来。不过文件中的数据默认是经过压缩的。使用上文推断出来的 `-nocompress` 参数（bu 确实支持这一参数，但在 adb backup 的文档中却没有提到过它），你可以生成一个存储的数据没有被压缩过的备份文件（见输出结果 3-12）。

```
morpheus@Forge (~) % adb backup -nocompress -all
# UI is displayed on device...
Now unlock your device and confirm the backup operation.
morpheus@Forge (~) % ls -l backup.ab
-rw-r----- 1 morpheus staff 17158168 Jan 1 23:37 backup.ab
morpheus@Forge (~) % head -6 backup.ab
ANDROID BACKUP      # MAGIC
3                   # Version (3 = L)
0                   # Compression (0 = False)
none                # Encryption (none)
apps/android/_manifest000600 0175001750000000036240010767 0ustar001
android
..
```

输出结果 3-12 创建一个未经压缩的备份文件，并查看其头部中存放的数据

备份文件头部中各个字段的含义我们还是很清楚的。但是备份文件中的具体内容又是怎么存储的呢？第一行数据看上去好像是元数据，所以我们试试把备份文件的头部去掉，然后再用 `file(1)` 命令试试运气。

```
# The header was four lines long, so start as of line 5...
morpheus@Forge (~) % tail +5 backup.ab > a.ab
# Attempt to auto identify the file..
morpheus@Forge (~) % file a.ab
a.ab: POSIX tar archive
# Check file contents:
morpheus@Forge (~) % tar tvtf a.ab | more
-rw----- 0 1000 1000      1940 Dec 31  1969 apps/android/_manifest
-rw----- 0 1000 1000          99 Jan  1 21:29 apps/android/r/wallpaper_info.xml
-rw----- 0 1000 1000      1961 Dec 31  1969 apps/com.android.browser.provider/_manifest
...
```

输出结果 3-13 去掉 Android 备份文件的头部

所以我们看到 Android 备份文件内部存放的也只不过是一个 UNIX 系统标准的 tar 文件，可以用 `tar` 命令把它解压出来。

如果你用口令保护备份文件，备份文件的头部会变得更大、更复杂。输出结果 3-14 显示的是一个经过压缩的且用口令“password”保护起来的 Android 备份文件。

```
# Note this time, no nocompress implies compress by default
morpheus@Forge (~) % adb backup -all
# UI is displayed on device... enter "password"
Now unlock your device and confirm the backup operation.
# File is significantly smaller this time
morpheus@Forge (~) % ls -l backup.ab
-rw-r----- 1 morpheus staff 7518645 Jan 1 23:50 backup.ab
morpheus@Forge (~) % head -9 backup.ab
ANDROID BACKUP
3
1      # This time, compressed
AES-256 # Encryption algorithm
FBAEB6CF..# 128 hex digits = 512-bit salt
98A4BF42..# 128 hex digits = 512-bit master key checksum
10000    # Number of PBKDF2 key derivations
0B0D638F9856C5D4F040399AB28A0C5F # Random IV (32 hex digits = 128bit)
E8AD4E9948F356E15A1E41AA265660.. # 192 hex digits = 768 bit Master key blob
```

输出结果 3-14 检查一个有口令保护的备份文件中的数据

监视备份操作

BackupManagerService 把它的配置存放在以下两个主要的位置上：

- 系统安全设置（The system secure settings）——这是所有 Android 框架服务通用的存放配置的地方，它可以通过 Settings 类访问。BackupManager 可以定义如表 3-10 所示（在 Settings 类中会把这些字符串转换成对应的常量）的设置。

表 3-10 控制备份行为的设置

设 置	用 途
backup_enabled	备份是否启用。等价于 bmgr enable 命令
backup_transport	默认的 transport。由 bmgr transport 命令设置
backup_provisioned	是否允许（provisioned）备份。对于被管理设备来说非常有用
backup_auto_restore	应用的数据是否可以自动恢复

- /data/backup 目录——其中记录了所有的 transports（存在一个子目录中）和备份队列。

在正常情况下，你不必亲自切换到这个目录中去进行设置，而是只要使用 bmgr（或者 settings）就能完成设置，也可以用 dumpsys backup 获取备份队列的详细信息，输出结果 3-15 给出的就是加了注释的 dumpsys backup 的执行结果。

```

shell@flounder:/ $ dumpsys backup
Backup Manager is disabled / provisioned / not pending init
Auto-restore is enabled
Last backup pass started: 0 (now = 1420171109885)
  next scheduled: 0
# List of transports. Google cloud is default, but requires account
Available transports:
  * com.google.android.backup/.BackupTransportServ
    destination: Need to set the backup account
    intent: Intent { act=com.google.android.backup.SetBackupAccountActivity }
  android/com.android.internal.backup.LocalTransport
    destination: Backing up to debug-only private cache
    intent: null
Pending init: 0
# List of applications that can request backup, sorted by AID:
Participants:
  uid: 1000
    com.android.providers.settings
    android
  uid: 1027
    com.android.nfc
...
# Ancestral refers to full backups, which serve as a point of
# departure for incremental backup/restore operations
Ancestral packages: none
Ever backed up: 0
Pending key/value backup: 13
  BackupRequest{pkg=com.google.android.gm}
  BackupRequest{pkg=com.google.android.talk}
...
Full backup queue:47
# Last backup : package name
  0 : com.android.providers.downloads.ui
  0 : com.android.externalstorage
  0 : com.google.android.nfcprovision
  ..

```

输出结果 3-15 使用 dumpsys backup 命令显示备份状态

## 实验：深度探索备份机制

为了更好地了解 Android 的备份机制，我们需要对/data/backup 目录做一番探索。这个目录是 BackupManagerService 记录它的元数据的地方。如果拥有 root 权限，你可以得到类似输出结果 3-16 的结果。

```

root@flounder:/data/backup # ls -l
drwx----- system system ... com.android.internal.backup.LocalTransport
drwx----- system system ... com.google.android.bac
-rw----- system system 1881 ... fb-schedule
drwx----- system system ... pending

```

输出结果 3-16 /data/backup 目录

要找出默认的 transport 是非常简单的，你既可以调用 bmgr 脚本，也可以直接去系统安全设置那里查询相应的值（见输出结果 3-17）。



```
root@flounder:/data/backup # bmgr list transports
* com.google.android.backup/.BackupTransportService
  android/com.android.internal.backup.LocalTransport
root@flounder:/data/backup # settings get secure_backup_transport
com.google.android.backup/.BackupTransportService
```

输出结果 3-17 找出默认的 transport

备份队列是记录在内存里的，但为了在备份服务崩溃时仍能恢复相关信息，它还会被写到 pending 目录中形如 journal-xxxx.tmp 的临时文件中。该文件记录的是一长串需要备份的包的包名。由于包名被处理为一个字符串，该字符串之前被冠以一个表示包名长度的字节，字符串又是用一个 NULL 表示结尾的，所以这个文件的内容看上去一般应该是输出结果 3-18 所示的样子。

```
root@flounder:/data/backup # cat -tv pending/journal-168056423.tmp
^@$com.android.providers.userdictionary^@'com.google.android.googlequicksearchbox^@
com.google.android.marvin.talkback^@$com.google.android.inputmethod.latin^@^Ucom.
google.android.gm^@^Ocom.android.nfc^@^Scom.android.vending^@^Gandroid^@^Wcom.google.
android.talk^@^_com.android.sharedstoragebackup^@)com.google.android.apps.genie.
geniewidget^@^[com.google.android.calendar^@^com.android.providers.settingsroot^@
```

输出结果 3-18 记录备份队列的 journal 临时文件中的内容

最后，fb-schedule 文件记录了所有已安装的包中，哪些是可以进行备份的包（也就是那些在 manifest 中申明过 BackupAgent 的应用，详见第 2 本的讨论，在 Android Developer 网站<sup>[11]</sup>中也有详细的文档）。这个文件的格式和 journal 的格式非常相似（尽管这个文件多了几个字段），而且它是 dumsys 获取信息的地方（这一点非常有用，因为你无须 root 权限就能使用它）。

## 3.5 系统重置（recovery）和升级

系统的重置（recovery）和升级的过程十分相似：系统都要用另一套代码去启动。用这套代码不会加载完整的操作系统用户界面，而是只加载一个最小的配置，用一个特定的二进制可执行文件/sbin/recovery 来执行我们目前正在讨论的这个启动过程。

不论是系统升级还是重置，一般都是在系统完全启动起来之后在图形界面里开始操作的。当然，也可以用 adb reboot recovery 命令或者通过 fastboot 让手机进入 recovery 模式。在通过图形界面开始系统升级或重置时，android.os.RecoverySystem 类提供了一个在用户发出升级请求时下载和验证升级包的框架（framework）。升级包必须经过数据签名，而且该签名要能被（手机中的）/system/etc/security/otacerts.zip 这个 keystore 中的证书验证为有效。只有签名被验证通过了，升级包才会被复制到/cache 分区中去。这也就是为什么在 Amazon 的 Kindle 阅读器（这款

设备会强制进行系统自动升级，而一旦升级了系统，之前对系统做的 root 当然也就作废了)中，删掉文件 `otacerts.zip` 就能阻止设备升级的原因。另一种(阻止设备升级的)有效的措施是把 `/cache` 分区的权限设为 `chmod root` 或者 `chmod 755`。

`android.os.RecoverySystem` 这个类也可以在 `recovery` 过程中传递参数。要传递的参数会被写入 `/cache/recovery/command` 文件，然后这个类会去重启系统，并在重启过程中传一个参数给 `Boot Loader`，让它从 `recovery` 分区而不是从 `boot` 分区启动。记得在之前的讨论中我们讲过，`recovery` 分区和 `boot` 分区中的内容差不多，都存有 `RAM disk` 镜像。只不过在从 `recovery` 分区启动时，加载的是 `/sbin/recovery`，而不是完整的 `Android` 操作系统。`RecoverySystem` 向 `/sbin/recovery` 传递命令的过程如图 3-7 所示。

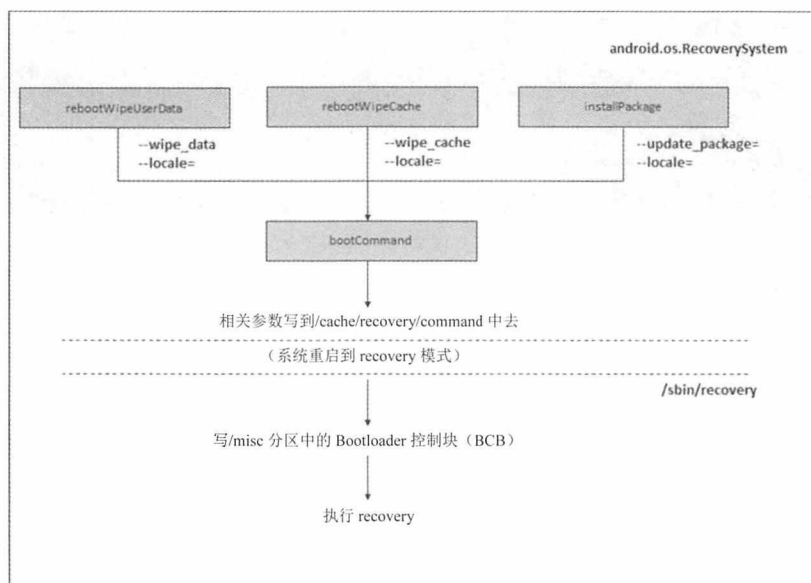


图 3-7 `android.os.RecoverySystem` 与 `/sbin/recovery` 之间的交互

`/sbin/recovery` 会从它自己的命令行中获得它的参数——如果有的话。如果没有参数的话，它还会去 `misc` 分区寻找 `BCB` (Boot Loader 控制块, Boot Loader Control Block)。如果找不到这个分区，或者分区里的内容没法解析，该可执行文件会转而使用 `/cache/recovery/command`。`Android` 运行时不会直接与 `BCB` 进行交互，它是留给 `/sbin/recovery` 保存所有给它的参数的。这样可以保证一旦出现了意外的中断，重启之后依旧会进入 `recovery` 模式，并再次执行之前失败的动作。

二进制可执行文件 `/sbin/recovery` 是肯定可以加载的，因为它是 `RAM disk` 的一部分，和 `/system` 一点关系都没有。注意，这一点很重要，因为 `/system` 可能会处于一种完全无法启动的

状态。然后，当内核初始化完毕，且加载了/init 之后，recovery（或许还有 adbd）也许就是除/init 之外的唯一一个被执行的进程了。接着/sbin/recovery 会去读取/cache/recovery/ command 文件中留给它的参数（如图 3-7 所示），并根据这些参数执行相关的操作。/cache/recovery/command 中可以使用的参数如表 3-11 所示。

表 3-11 /sbin/recovery 所能理解的参数

参 数	用 途
--wipe_cache	擦掉/cache 分区，并重启
--wipe_data	擦掉/data 分区中的所有用户数据——也就是“恢复出厂设置”。在手机的整个生命周期里，/system 通常都是以只读方式 mount 的，因此极少会有其中的数据遭到破坏的情况发生。把设备恢复到出厂设置也就意味着格式化/data 分区。这不光会把用户自己的所有数据都擦掉，同时也会清除掉那些可能影响设备正常启动的文件。使用这一参数也意味着要去执行 wipe_cache 参数的相关操作
--update_package	指定升级系统所用的 OTA 升级包的存储路径。OTA 包会在下一节中讨论
--locale	指定所使用的语言，该设置会被写入/cache/recovery/last_locale 文件
--send_intent	将被写入/cache/recovery/intent 文件的 intent 的名字
--show_text	显示文本消息
--just_exit	不执行任何操作直接退出

在整个过程中，不断向用户输出信息，并保持屏幕点亮是非常重要的。所以 recovery 也要使用 minui（顾名思义，它就是一个提供基本 GUI 功能的库）来完成相关操作。我们会在第 2 本中更详细地讨论这个库。

### OTA（Over-The-Air）升级包

厂商、运营商（甚至是谷歌自己）会隔三差五地以 OTA（Over-The-Air）的形式推送 Android 操作系统的升级包。因为是通过无线网络推送的，所以升级包必须做得尽可能小。根据被升级 Android 系统的 build 版本不同，所对应的 OTA 增量升级包也是不一样的。

Android 的 OTA 升级文件是压缩存放在一个经过数字签名的 zip 文件中的（从技术上讲，称之为 JAR 包更合适些，因为其中含有一个 META-INF/子目录）。其中的内容有：

- 多个补丁文件——这些补丁文件都是 bsdiff(1) 格式的。这种文件格式实际上就是一串记录，每个要修补的地方对应一条记录，其中记录的要修补的位置在文件内的偏移量、要修补的长度以及要在这个位置上插入或删除的数据。标准的补丁文件的文件名就是要修补的文件的文件名，后面再跟上一个扩展名“.p”。



- 一个补丁程序——（它的文件名通常为 `update-binary`）这个程序可以解析各个补丁文件，并去补丁脚本指定的路径中找到被修补文件，并升级之。
- 一个补丁脚本——（它的文件名通常为 `updater-script`）该脚本会多次去调用补丁程序中的函数（每当有一个要升级的文件时就要调用一次），并且在升级操作之前/之后都计算并验证一下被升级的文件的 Hash。
- 需要增加的文件——这些文件可以是系统中要新增的文件，也可以是那些由于要修改的地方实在是太多，以至于用 `bsdiff(1)` 格式写出来，比正常文件还要大的需升级的文件。
- 一个 **metadata** 文件——其中记录了 `post-build`、`post-timestamp`、`pre-build` 和 `pre-device` 属性的值，这些值表示了升级前后设备的相关属性（property）。
- 一个 **otacert** 文件——这是一个 PEM 格式的证书。它可以被验证是不是 `/system/etc/security/otacerts.zip` 中某个的证书，也可以被导入到 `/system/etc/security/otacerts.zip` 中。

注意，厂商可以不受限制地增加或修改 OTA 中的任意组件。Amazon 的 Kindle 的升级包就是一个很好的例子——这些升级包不光会去升级 `/system` 分区中的文件，还会额外地去升级固件镜像，甚至包括一些不会被系统 mount 的分区。

代码清单 3-4 中给出的就是谷歌推送给 Nexus 5 设备的 KitKat 的 OTA 升级包中的内容。

代码清单 3-4 Nexus 5 的某个 OTA 升级包中的内容

```
morpheus@Forge (/tmp)$ unzip -l 537....740.signed-hammerhead-KOT49H-from-KOT49E.537367d5.zip
Archive: 537367d588afe31301268d0ace7e60725d5f6740.signed-hammerhead-KOT49H-from-KOT49E.537367d5.zip
signed by SignApk
  length    date       time        name
  -----
    203     09-03-13   11:53      META-INF/com/android/metadata
   275280   09-03-13   11:53      META-INF/com/google/android/update-binary
   132207   09-03-13   11:53      META-INF/com/google/android/updater-script
    2045     09-03-13   11:53      patch/system/app/BasicDreams.apk.p
    ...
    575     09-03-13   11:53      recovery/etc/install-recovery.sh
   100345   09-03-13   11:53      recovery/recovery-from-boot.p
   487562   09-03-13   11:53      system/framework/telephony-common.jar
    23252   09-03-13   11:53      META-INF/MANIFEST.MF
    23305   09-03-13   11:53      META-INF/CERT.SF
    1346    09-03-13   11:53      META-INF/CERT.RSA
    1229    09-03-13   11:53      META-INF/com/android/otacert
```

## OTA 升级过程

在用 `--update_package` 参数启动系统时，`recovery` 程序就会去调用 `install_package()` 函数，该函数会去加载参数中指定的升级包，并在其中寻找二进制可执行文件 `update-binary`。这个二进制可执行文件的名称是事先在 `META-INF/com/google/android/update-binary` 中用 `#defined` 语句在变量 `ASSUMED_UPDATE_BINARY_NAME` 中规定死了的。如果它被找到了，那么 `recovery` 程序就会运行它，`update-binary` 又会以类似在 Linux 中执行 shell 脚本的方式，去执行 `updater-script`

脚本。

标准的 update-binary 的源码可以在 Android 源码树中找到（在 bootable/recovery 目录中）。大多数 OTA 升级包中使用的 update-binary 都是通过修改这个源码得来的，因为一般是鼓励厂商修改源码实现特定功能的。除了修改它之外，厂商也可以很方便地在其中添加一些被称为“设备扩展”（device extension）的其他的库（library）。只要把相应的库写入 Android.mk 文件的 TARGET\_RECOVERY\_UPDATER\_LIBS 变量中，并在每个设备扩展库中提供一个 Register\_libname 函数即可完成设备扩展库的添加。

看一下 update-binary 的源码，你会在 RegisterInstallFunctions()函数的实现中发现所有可以在 update-script 中使用的函数的函数名。你可以用 grep(1)命令搜索“RegisterFunction”的方式找到它们（见代码清单 3-5）。

代码清单 3-5 update-binary 的源码中所有调用 RegisterInstallFunctions()的行

```
# Block based updates introduced in L
blocking.c: RegisterFunction("block_image_update", BlockImageUpdateFn);
blocking.c: RegisterFunction("range_shal", RangeShalFn);
# Block based verify added in M
blocking.c: RegisterFunction("block_image_verify", BlockImageVerifyFn);
# Rest of functions in updater core
install.c: RegisterFunction("mount", MountFn);
install.c: RegisterFunction("is_mounted", IsMountedFn);
install.c: RegisterFunction("unmount", UnmountFn);
install.c: RegisterFunction("format", FormatFn);
install.c: RegisterFunction("show_progress", ShowProgressFn);
install.c: RegisterFunction("set_progress", SetProgressFn);
install.c: RegisterFunction("delete", DeleteFn);
install.c: RegisterFunction("delete_recursive", DeleteFn);
install.c: RegisterFunction("package_extract_dir", PackageExtractDirFn);
install.c: RegisterFunction("package_extract_file", PackageExtractFileFn);
install.c: RegisterFunction("symlink", SymlinkFn);
install.c: RegisterFunction("set_metadata", SetMetadataFn);
install.c: RegisterFunction("set_metadata_recursive", SetMetadataFn);
install.c: RegisterFunction("getprop", GetPropFn);
install.c: RegisterFunction("file_getprop", FileGetPropFn);
install.c: RegisterFunction("write_raw_image", WriteRawImageFn);
install.c: RegisterFunction("apply_patch", ApplyPatchFn);
install.c: RegisterFunction("apply_patch_check", ApplyPatchCheckFn);
install.c: RegisterFunction("apply_patch_space", ApplyPatchSpaceFn);
install.c: RegisterFunction("wipe_block_device", WipeBlockDeviceFn);
install.c: RegisterFunction("read_file", ReadFileFn);
install.c: RegisterFunction("shal_check", ShalCheckFn);
install.c: RegisterFunction("rename", RenameFn);
install.c: RegisterFunction("wipe_cache", WipeCacheFn);
install.c: RegisterFunction("ui_print", UIPrintFn);
install.c: RegisterFunction("run_program", RunProgramFn);
# Added in L
install.c: RegisterFunction("reboot_now", RebootNowFn);
install.c: RegisterFunction("get_stage", GetStageFn);
install.c: RegisterFunction("set_stage", SetStageFn);
install.c: RegisterFunction("enable_reboot", EnableRebootFn);
install.c: RegisterFunction("tune2fs", Tune2FsFn);
```

在 Android 开发者网站中的“OTA Updates: Inside OTA Packages”网页<sup>[12]</sup>上，可以找到关于这些函数的相当详细的文档以及使用说明。

代码清单 3-6 中给出的是一个我加了注释的 updater-script 脚本的例子。这个 updater-script



是从一台三星手机的 Android 4.4.2 OTA 升级包中提取出来的。这段代码中不光有代码清单 3-5 中给出的函数，还有使用它们时所用的参数。

代码清单 3-6 一个带注释的 updater-script 的例子

```
# Mount partitions, r/w, for purposes of overwriting and patching
mount("ext4", "EMMC", "/dev/block/platform/msm_sdcc.1/by-name/system", "/system");
mount("ext4", "EMMC", "/dev/block/platform/msm_sdcc.1/by-name/hidden", "/preload");
# Pre-verification: note the use of short circuit evaluation to either pass
# the check, or proceed to potentially abort update

file_getprop("/system/build.prop", "ro.build.fingerprint")
    == "samsung/jfltetmo/jfltetmo:4.3/JSS15J/M919UVUEMK2:user/release-key
    ... ||
    abort("Package expects build fingerprint of .....");
# getprop makes sure the device's built or product version match
assert(getprop("ro.product.device") == "jfltetmo" ||
    getprop("ro.build.product") == "jfltetmo" ||
    getprop("ro.product.device") == "jfltetmo" ||
    getprop("ro.build.product") == "jfltetmo");
# ui_print displays a text message for the user, and show progress displays the bar
ui_print("Verifying current system...");
show_progress(0.100000, 0);
# apply_patch check applies the corresponding .p file and
# validates the SHA-1 before and after
apply_patch_check("/system/framework/framework-res.apk",
    "384b5aa8862c18a53fbfa8f8e3789d7edc4561ab",
    "57d8eca2e983fc1031841d418e759246c9885245")
    || abort("/system/framework/framework-res.apk" has unexpected contents.");
..
set_progress(0.078730);
# apply rename check is as apply patch check, for an mv.
# Assert verifies return value
assert(apply_rename_check("/system/app/SecGallery2013.apk",
    "/system/priv-app/SecGallery2013.apk",
    "8e0eb424e636a06733c3e08cdf29361d97d23929",
    "66d8f058e1638124f618a90773aa4e8bad7765d8"));

# apply_patch_space returns false if space in bytes is less than quoted value
apply_patch_space(70620592) || abort("Not enough free space on /system to apply patch");
assert(getprop("ro.secure")=="1");
..
# delete recursive removes files or directories
delete_recursive("/system/app/CABLSERVICE.apk",
    ..
    symlink("toolbox", "/system/bin/mkswap", "/system/bin/readlink",
        "/system/bin/swapoff",
        "/system/bin/swapon");
#...
# set_perm recursive sets uid, gid and permissions on files or directories*
set_perm_recursive(0, 2000, 0755, 0755, "/system/xbin");
ui_print("Update tz.mbn ota");
assert(write_backup_fota("/dev/block/mmcblk0p8", "/dev/block/mmcblk0p22"));
assert(package_extract_file("tz.mbn", "/dev/block/mmcblk0p8"));
assert(erase_backup_fota("/dev/block/mmcblk0p22"));
..
show_progress(1.000000, 0);
unmount("/preload");
unmount("/system");
```

## 制作你自己的 ROM

在学习了本章上述这些知识之后，你现在已经掌握了定制一套你自己的固件镜像（ROM）所需的所有知识了。注意，作为使用定制 ROM 的先决条件，手机的 Boot Loader 必须已经解锁



了，否则是没办法把自制的 ROM 刷到手机里的。

在使用定制的 ROM 时，通常会冒以下风险：

- 从不可信的网站那里下载下来的 ROM 中很可能带有恶意软件，特别是间谍软件。
- 使用定制的 ROM 可能会（当然也有可能不会）使运营商（或者厂商）拒绝保修你的手机。换句话说，如果 Boot Loader 没有被修改，那么这一过程将是可逆的。有些手机/平板电脑不会保留 Boot Loader 被修改的记录，在这种情况下，甚至有可能再把 Boot Loader 加锁回去，这样就不会留下任何修改过的痕迹了。
- 错误地安装一个定制的 ROM，有可能使设备不能启动（也就是俗话说的“变砖头”了）。变砖的手机到底能不能被救回来，取决于你这个 ROM 究竟修改了哪些分区。
  - 如果只修改了/data 分区，但 OS 还是能够从/system 分区启动的。在最坏的情况下，也只需要恢复出厂设置（即擦除/data 分区）就能把设备修好。
  - 如果/data 分区和/system 分区都被修改了，内核和 ramdisk 这时应该还是能够（从 boot 分区）正常启动的，那么系统应该是能进入 rescue 模式的。
  - 如果 boot 分区（内核+initramfs）也已经被修改了，内核和 ramdisk 可能就不能启动了，但是这时 Boot Loader 应该还是能启动的，所以还是有望通过 fastboot（或者其他定制的协议）重写 boot 分区，使得设备能够再次重新启动。
  - 如果 Boot Loader 本身也需要被修改，那么你对 Boot Loader 所做的修改确实有可能让手机无法启动，这时手机就真有可能永远变成砖头了。

了解了上述这些危害之后，我们就明白在接下来的操作中，适当地加点小心总是不会有错的。大多数的 Boot Loader 允许你在不修改现有的任何一个分区中数据的情况下，从另一个镜像启动（通过 fastboot）。这样做的目的就是提供了一个安全的测试环境。接下来我们就来讲讲定制一个 ROM 的方法。

### 在已有文件系统中加入新的文件

如果你的手机已经 root 了，那么在一个已有的文件系统中加入新的文件，实在是件轻而易举的事。你首先要确保文件系统处于可写状态，如果没有的话，就要用可写模式重新 mount 一下这个分区，然后就只需把新的文件复制到这个文件系统里去就行了。由于 adb 通常不是以 root 权限运行的（尽管默认是这样的，但我们也可以改），所以复制文件这一操作一般要分成两步完成：先把文件（通过 adb push）放到一个（adb）可以写的目录中（比方说，/data/local/tmp 就是个不错的选择），然后再用 cp 命令 [因为可能会涉及链接（link），在 toolbox 中，mv 命令一般都无法正常工作]，把这些文件复制到它们真正的目标文件夹中去。



这个方法要做的操作比我们待会儿还会介绍的一种方法要多，但是一般来说，这个方法是安全的。如果你只是要添加新的文件，且不需要覆盖系统中已有的文件，用这一方法对系统造成破坏的可能性是非常小的。在我们添加 `property` 文件或者其他会改变系统行为的文件时，应该遵循风险最小化这一原则。

## 修改 initramfs

当你要修改 `/data` 或 `/system` 分区之外的文件时，你修改的其实就是 `initramfs`。我们回忆一下先前的讨论：先要把 `initramfs` 用作 `root` 文件系统，然后内核才能把 `/data` 和 `/system` 文件系统 mount 到相应的位置上去。在 `initramfs` 中有一个 `/init.rc` 文件，这个文件中记录的是 `/init` 在启动系统时要执行的指令。所以修改 `initramfs`，在其中插入相应的指令，是 `root` 手机的一个不错的方法。下面这个实验中演示了这一攻击方法。

## 实验：重新制作一个 `initramfs` 镜像，并把它作为 `bootimg` 写入手机

本章的上文中有个实验，演示了如何解压并提取 `initramfs` 中的内容，而重新把 `initramfs` 打包压缩到 `gzip` 镜像中，实际上就是这一操作的逆过程。即用 `-o` (output, 输出) 参数而不是 `-i` (input, 输入) 参数运行 `cpio`，然后再执行 `gzip` 完成压缩就行了（见输出结果 3-19）。你也可以使用 AOSP 的 `/system/core/cpio` 目录中的 `mkbootfs` 工具来完成这一工作。

```
morpheus@Forge (~/.Android/Book/tmp) % find . | xargs cpio -ovd | gzip > output.gz
```

输出结果 3-19 创建一个压缩了的 `initramfs`

注意，`initramfs` 并不是单独提供的，它总是和内核打包在一起的。所以，你得用 `make` 参数运行 `imgtool`（或者也可以使用 AOSP 的 `mkbootimg` 工具），把 `initramfs` 和内核打包在一起。只有这样，这两个东西待会儿才能被刷到 `boot` 分区里。我假定你已经在之前的实验里手工拿到了 `kernel`（内核）文件——一般没有必要去修改它。这一操作如输出结果 3-20 所示。

```
morpheus@Forge (~/.Android/Book/tmp)% mkbootimg --kernel kernel \
--ramdisk output.gz --output bootimg.img
# Alternatively, with imgtool:
morpheus@Forge (~/.Android/Book/tmp)% imgtool make bootimg.img kernel output.gz
```

输出结果 3-20 用内核和经过压缩的 `initramfs` 创建一个 `bootimg` 镜像

然后，我们就可以用 `fastboot flash` 命令（假设你的 `Boot Loader` 已经解锁了）把它写到你的手机中。或者你也可以用 `dd` 命令（必须非常小心！）完成这一任务（假设你的手机已经 `root` 了）。下面的操作中，我们是用 `by-name` 符号链接得到 `boot` 分区的具体位置的（见输出结果 3-21）。注意，在不同的设备上，这些命令（特别是 `grep boot /proc/partitions`）的执行结果应该是不一



样的（而且你要特别小心，别把 about 分区或 hboot 分区和 boot 分区搞混了）。

```
morpheus@Forge (~/.Android/Book/tmp)% adb push bootimg.img /data/local/tmp
morpheus@Forge (~/.Android/Book/tmp)% adb shell
shell@Android /$ grep boot /proc/partitions
..
..
shell@Android /$ su
root@Android /# dd if=/data/local/tmp/bootimg.img \
of=/dev/block/platform/msm-sdcc.1/by-name/boot
```

输出结果 3-21 用 dd 命令重写 boot 分区

为了增加安全系数，你可以重复之前那个实验，解压 bootimg 镜像，验证一下，然后再把它打包回去，并把你打包得到的结果写回到设备中。此外，保留一份修改前的 bootimg 镜像总是个不错的主意，这样，在你遇到问题的时候，就能很方便地把它再（通过 fastboot）刷回到手机上去了。

## 重刷整个分区

有的时候，重刷整个分区比往分区里添加文件更方便。通常在两种情况下会出现这一现象：其一，在定制过程中要增删许多文件（比如说要精简厂商安装的冗余软件）；其二，需要修改不能 mount 的分区里的内容。无论是哪种情况，首先要做的就是用 dd(1) 程序直接把这个分区对应的块设备中的所有数据二进制逐字节地复制到一个镜像文件中，或者你也可以用 adb push 命令把块设备中的所有数据二进制逐字节地直接复制到电脑上（的某个镜像文件中）。这个制作分区镜像的方法，从本质上来说，和我们在第 2 章中介绍的方法是一模一样的。当然你也可以像之前的实验那样，去修改已有的 Android 系统镜像。

## 实验：制作 Android 的/system 分区镜像

重新制作某个分区的镜像，是上述过程的逆过程。你首先要（在电脑上）用 mkextfs 命令创建一个新的文件系统，然后往里面写入你要添加的文件。在 unmount 这个文件系统时，对该文件系统的所有修改就会被写到镜像中。然后你把它复制到设备里，再用 dd(1) 程序（交换一下 if 和 of 这两个参数）把新的分区镜像刷到目标分区上（这步操作务必万分小心！）。此外，你也可以使用 fastboot 把镜像刷到目标分区上，但是如果出现了 fastboot 拒绝刷没有数字签名验证的分区镜像的情况，使用 dd(1) 或许是个更有效的方法。

在之前的实验里，我让你编译过 simg2img 程序，现在我们要反过来编译 img2simg 程序了。这样，你就能够先制作一个纯二进制（raw）格式的文件系统镜像，然后再用 img2simg 程序把它转换成谷歌使用的 simg 格式的镜像。如果你想要为你的手机中某个分区定制一个镜像，并把它刷到设备中的话，这个工具是非常有用的。你可以先从修改 Android 的分区镜像开始，往里





面加入更多的文件（不知为什么，我脑子里立马飘过了 SetUID “su”），然后用 dd(1)把已经 mount 上来的文件系统逐字节地复制成一个纯二进制（raw）格式的镜像。这时，只要用 img2simg 程序就能由这个镜像再生成一个 simg 格式的镜像，接下来就能用 fastboot 把 simg 格式的镜像刷到你的手机里了（这将在本章的稍后部分中予以讨论<sup>1</sup>）。

## 制作 ROM 时可用的网上资源

本书中关于定制 ROM 的讨论为你进一步的深造打下了一个坚实的基础——如果你确实想进入这个激动人心，当然还有那么点危险的领域的话。幸运的是，这方面确实有一些专业的站点，它们是无价的知识宝库。其中既有其他网友定制和测试 ROM 的经验，也有把手机刷成砖头的惨痛教训。

### XDA-Developers

对于 Android 高级用户来说，这是最重要的资源站点！XDA-Developers<sup>[13a]</sup>网站上存有海量的关于 Android 的所有信息。事实上，在该网站的讨论区<sup>[13b]</sup>中的分论坛和高质量的讨论中涵盖了市面上所有 Android 设备。root 和 recovery 工具一般也会最先拿到这里来发布，而且在这个拥有近 6 百万用户（其中还有在定制 ROM 的世界里最大名鼎鼎的人物）的社区里，你的所有问题几乎总是能得到帮助——实在太傻的问题除外。

### Cyanogen、AOKP 等

由于 Android 本身是开源的，谷歌在很长的一段时间里都没有对定制 ROM 或者修改 Android 操作系统的团队有过什么动作。事实上，AOKP（Android Open Kang Project）、CyanogenMod 等好几个（开源）项目都在干这方面的活。CyanogenMod 甚至已经能通过修改厂商提供的内核和系统镜像的方式，为市面上各种（Android）手机提供一站式的专用刷机 ROM 包了。CyanogenMod 的开发人员会用各种方式改进 Android 系统，这可不仅仅是 root（root 只不过是制作 ROM 时顺手搞定的），在大多数情况下，ROM 中还会整合进最新版的 Android 系统补丁和新的功能，这使得在厂商提供官方升级包之前，CyanogenMod 的 ROM 更加好用。

成就 CyanogenMod 的原因是：厂商必须提供它们的 Android 系统的内核源码。再加上 AOSP 本身就是开源的，这使得其他许多开发者能够为大幅改进厂商发布的系统提供帮助，也使得其他人能跳出厂商（通常是极为拖沓的）的更新周期，发布自己的 ROM。因此，对给定型号的手机进行的改进，既可以是以厂商内核源码的形式发布的，也可以以二进制可执行文件〔即系统

---

1 原文如此，可是你看看，本章还有多少下文？不过这一技巧在上文 fastboot 那一节中应该已经介绍了。——译者注



模块 (module)] 的形式给出。不过在后一种情况下, 需要重新编译内核, 以确保兼容其他厂商提供的模块。此外, 还可以很方便地删掉厂商捆绑在系统中的那些应用, 或者把其他的应用捆绑在 ROM 里。

在大多数情况下, 与其自己定制一个 ROM, 还不如直接使用 CyanogenMod 提供的已经做好了 ROM。因为这能使你规避因为定制 ROM 过程中的某个失误而把自己的手机刷成砖头的风险。CyanogenMod 甚至已经开始提供它们自己的 Boot Loader 了, 整个定制和安装 ROM 的过程已经简化成点击几下鼠标就能搞定的事了。我强烈建议对此有兴趣的读者去访问 CyanogenMod 的官网<sup>[14a]</sup>, 以获取更多的信息。特别是 CyanogenMod 还有一个维基网页<sup>1</sup>, 此外它们的开发者学习中心 (Development Learning Center)<sup>[14b]</sup>也是个特别好的参考网站。

CyanogenMod 甚至都成了另一种 Android 标准, 有一些厂商 (特别是一加手机) 甚至把它作为默认的 Android 系统。现在 Cyanogen 已经从微软、Foxconn 等公司拿到了超过一亿美元的投资, 完美地创业起步了。

## Team-Win Recovery 项目

Team-Win Recovery 项目<sup>[15]</sup>是个定制的 recovery 镜像, 它由一群 “好胜的 Android 爱好者” 创建。和其他的 recovery 镜像一样, 它是由一个打包了内核和 RAM disk 并用一定的格式存储的 bootimg 镜像组成的。其中的内核就是个普通的内核, 但 RAM disk 中存放的却是个功能完备的 recovery 程序, 它能显示出一个挺有个性特色的图形用户界面, 提供备份和恢复 (restore) 功能, 支持 ExFAT、F2Fs 以及 Ext 系列文件系统。另外其中还带有一个 ChainFire 的 SuperSU——这是一个被广泛使用的 root 工具, 它使得 Team-Win 的 recovery 能够 root bootloader 已经解锁了的手机。对于高级用户来说, TWRP<sup>2</sup>这个镜像让你能快速获得 busybox 的二进制可执行文件和编译好的 Bionic 库, 当然也能控制在屏幕上显示的信息, 比如在 `ls -l` 命令的输出结果中显示 AID。

TWRP GUI 层上的显示是由 `res/ui.xml` 文件 (这个文件可以在 Team-Win 的 GitHub 代码仓库<sup>[16]</sup>中下载, 你也可以自行修改它) 控制的。`res/`目录中还存有字体文件 (Font) 和要显示在屏幕上的图像 (以 PNG 文件的形式提供)。TWRP 中的 recovery 程序比正常的 `/sbin/recovery` 程序要大, 它用的是一个改过的 AOSP MinUI 库 (`libminui.so`)。这个被改过的 MinUI 库能显示一个简单的 GUI 界面, 并支持触摸操作 [相比之下, 在标准的 recovery 程序中, 用户必须要用手机上的物理键 (音量增大/减小键和电源键) 才能进行操作]。但是为了让镜像保持紧凑, 所有

---

1 [https://wiki.cyanogenmod.org/w/Main\\_Page](https://wiki.cyanogenmod.org/w/Main_Page)。——译者注

2 Team-Win Recovery Project 的缩写。——译者注



Android 运行时的特性以及服务都被精简了(不过由于系统启动过程的需要,还是要加载一个“准”/init 程序的),如图 3-8 所示。我们将在第 2 本中再来讨论 MinUI。

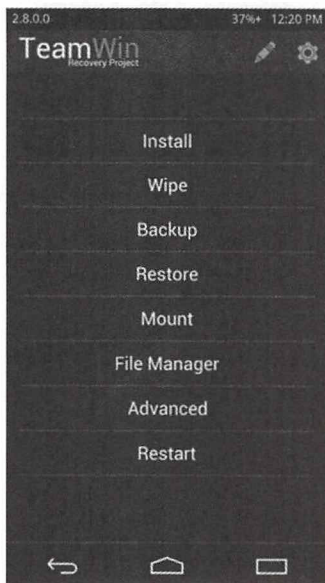


图 3-8 TWRP 2.8 (本书编写时的最新版本)的图形用户界面

## 本章小结

本章中讨论了 Android 的启动流程和它的生命周期。首先我们讨论了这一过程涉及的各种镜像的格式,然后一步一步地讨论了系统启动的操作步骤。我们还讨论了关机和重启操作,特别是“启动到 recovery 模式”的内部实现方式,接着还讨论了 recovery 模式的概念和/sbin/recovery 的作用。最后,我们还详细讨论了定制 ROM 或者修改系统镜像的方法。

你一定已经注意到:到目前为止,我们还没有讨论过用户模式下的启动过程,即在内核初始化完毕,启动了 PID 为 1 的进程(/init)并运行了各种 Android 原生服务之后,系统中又发生了些什么。在第 5 章中,我们将非常详细地讨论/init 的作用以及各种原生服务,而 framework 服务则要留待第 2 本再讨论。

## 参考文献

- [1] Companion Website, ImgTool source + binary: <http://NewAndroidBook.com/files/imgtool.tar>
- [2] Google Nexus Factory Image Repository: <https://developers.google.com/android/nexus/images/>
- [3] a. CodeAurora, LK: <https://www.codeaurora.org/cgit/quic/la/kernel/lk/tree/app/aboot>





- b. Google Source, LK: <https://android.googlesource.com/kernel/lk/+a9b07bbae16a0b1b6de07ec3a3e2005c99043757/>
- [4] a. Accuvant, Building a Nexus 4 UART Debug cable: <http://www.accuvant.com/blog/building-a-nexus-4-uart-debug-cable>  
b. OSDevNotes Blog, 64-Bit ARM Kernel Development demo: <http://osdevnotes.blogspot.com/2014/11/64-bit-arm-oskernelsystems-development.html>
- [5] Companion article, Disassembling ABoot <http://NewAndroidBook.com/Articles/about.html>
- [6] Android Documentation, Building Kernels: <http://source.android.com/source/building-kernels.html>
- [7] ePAPR DTB specification: [https://www.power.org/wp-content/uploads/2012/06/Power\\_ePAPR\\_APPROVED\\_v1.1.pdf](https://www.power.org/wp-content/uploads/2012/06/Power_ePAPR_APPROVED_v1.1.pdf)
- [8] Thomas Pettazoni, "Device Tree for Dummies": <http://elinux.org/images/a/a3/Elce2013-petazzoni-devicetree-for-dummies.pdf>
- [9] a. XDA-Developers, Discussion of Qualcomm leaked documents: <http://forum.xda-developers.com/showthread.php?t=1856327&page=1>  
b. Thread 24100141
- [10] Android Developer, utility: <http://developer.android.com/tools/help/bmgr.html>
- [11] Android Developer, Declaring Backup Agent in Manifest: <http://developer.android.com/guide/topics/data/backup.html#BackupManifest>
- [12] Android Source, "Inside OTA Packages": [http://source.android.com/devices/tech/ota/inside\\_packages.html](http://source.android.com/devices/tech/ota/inside_packages.html)
- [13] a. XDA Developers: <http://www.xda-developers.com>  
b. XDA Developers - Forums: <http://forum.xda-developers.com>
- [14] a. CyanogenMod: <http://www.cyanogenmod.org/>  
b. CyanogenMod Wiki: <http://wiki.cyanogenmod.org/w/Development>
- [15] Team-Win Recovery Project: <http://teamw.in/project/twrp2>
- [16] Team-Win GitHub Repository: <https://github.com/TeamWin/Team-Win-Recovery-Project/tree/jb-wip/gui/devices>



## 第 4 章

# init

所有的 UNIX 系统都有一个特殊的进程，它是内核启动完毕后，用户态中启动的第一个进程，它负责启动系统，并作为系统中其他进程的祖先进程。传统上，这个进程被称为 `init`，Android 也沿用了这个约定俗成的规矩。

不过 Android 中的 `init` 和 UNIX 或 Linux 中的 `init` 还是有很大不同的，其中最重要的不同之处在于：Android 中的 `init` 支持系统属性，并使用一些指定的 `rc` 文件（来规定它的行为）。在介绍完这两个特性之后，我们将拼凑出一幅 `init` 运行流程的全景图：它的初始化过程和在主循环（`run-loop`）中要做的操作。

另外，`init` 还扮演着其他的角色——它还要化身为 `ueventd` 和 `watchdogd`。这两个重要的核心服务也是由 `init` 这个二进制可执行文件来实现的——通过符号链接的方式加载。

### 4.1 `init` 的角色和任务

和大多数 UNIX 内核一样，Linux 内核会去寻找一个路径和文件名预先规定好的二进制可执行文件，并把它作为第一个用户态进程执行。在桌面 Linux 系统中，这个可执行文件一般是 `/sbin/init`，它会去读取 `/etc/inittab` 文件的内容，以获取所支持的“运行级”（`run-levels`）、运行时配置信息（单用户、多用户、网络文件系统等）、需随机启动的进程以及当用户按下 `Ctrl-Alt-Del` 组合键时该做出何种反应等信息。Android 也使用这样一个 `init` 程序，但是 Android 中的 `init` 程序和传统 Linux 中的 `init` 程序的相似之处也就仅止于这个文件名了，它们之间的区别如表 4-1 所示。



表 4-1 Android 的/init 与传统 UNIX 的/sbin/init 的对比

	Linux 的/sbin/init	Android 的/init
配置文件	/etc/inittab	/init.rc 以及它导入的任何文件 [ 通常是 init.hardware.rc 和 init.usb.rc ( 有时是 init.hardware.usb.rc ) ]
多种配置	支持: “运行级” (run-levels) 的概念 (0: 系统停机状态; 1: 单用户工作状态; 2-3: 多用户状态; ……)。每个“运行级”都会从 /etc/rcrunlevel.d2 那里加载脚本	没有“运行级”的概念, 但是通过触发器 (trigger) 和系统属性提供了配置选项
看门狗 (Watchdog) 功能	支持: 用 respawn 关键字定义过的守护进程会在退出时重启——除非该进程反复崩溃, 在这种情况下, 反复崩溃的进程会被挂起几分钟	支持: 服务默认是应该保持活跃的, 除非启动脚本中启动它时使用了 oneshot 参数。服务启动时还可以使用 critical 参数, 这会使系统在该服务无法重启时被强制重启
收容孤儿进程	支持: /sbin/init 会调用 wait4() 系统调用来获取 (孤儿进程的) 返回码, 并避免出现僵尸进程	支持: /init 注册了一个 SIGCHLD 信号的处理模块。SIGCHLD 信号是内核在子进程退出时自动发送的。大多数进程会默默地调用 wait(NULL) 清理掉已退出的子进程, 而不去管它的退出码是什么
系统属性	不支持: Linux 的/sbin/init 不支持“系统属性”这个概念	支持: /init 通过共享一块内存区域的方式, 让系统中的所有进程都能读取系统属性 (通过 getprop), 并通过一个名为“property_service”的 socket 让有权限的进程能够 (通过 setprop) 写相关的属性
分配 Socket	不支持: Linux 的 init 不会向子进程提供 socket, 这个功能是交给 inetd 的	支持: /init 会绑定一个 UNIX domain socket (从 L 版开始, 是 seqpacket socket) 提供给子进程, 子进程可以通过 android_get_control_socket 函数获取到它
触发操作	不支持: Linux 只支持非常特殊的触发操作, 比如 ctrl-alt-del 和 UPS 电源事件, 但是不允许任意的触发操作	支持: /init 可以在任何一个系统属性被修改时, 执行记录在 trigger 语句块中的, 由任何一个 (或某些) 用户预先写好的指令
处理 uevent 事件	不支持: Linux 依靠的是 hotplug 守护进程 (通常为 udevd)	/init 也会化身为 ueventd, 用专门的配置文件来指导其行为

1 系统默认运行级不能设为 0, 否则不能正常启动运行级。——译者注

2 /etc/rcrunlevel.d 中的 runlevel 应该被替换为对应的用数字表示的运行级。——译者注





/init 是个静态链接的二进制可执行文件。也就是说在编译时，它的所有依赖库都已经被合并到这个二进制可执行文件里去了。这样做是为了防止仅仅因为缺少某个库或者某个库被破坏而造成系统无法正常启动的情况发生。在/init 刚被执行时，只有和内核被一起打包放在 boot 分区上的 RAM disk（也就是 root 文件系统）被 mount 了上来，换句话说，系统中只有 / 和 /sbin。

从某种意义上说，Android 的 init 更像 iOS 的 launchd。不过后者的特色是提供了触发器（trigger）和套接字（socket）这两个特性，而 Android（的 init）的特点则是引入了系统属性（system property）。

### 系统属性

Android 的系统属性提供了一个可全局访问的配置设置仓库<sup>1</sup>。它们在形式和功能上都和各种 MIB 数组作为参数调用的 sysctl(2)有些类似，只不过是用户态中，由 init 实现罢了。与 init 相关的源码中的 property\_service.c 中的代码，会按表 4-2 中给出的顺序，从多个文件中加载属性。

表 4-2 Android 文件系统属性文件

文 件	内 容
/default.prop (PROP_PATH_RAMDISK_DEFAULT)	初始设置。注意，这个文件是 initramfs 的一部分，直接到设备的闪存分区上是找不到它的
/system/build.prop (PROP_PATH_SYSTEM_BUILD)	编译 Android 过程中产生的设置
/system/default.prop (PROP_PATH_SYSTEM_DEFAULT)	通常是厂商添加的设置
/data/local.prop (PROP_PATH_LOCAL_OVERRIDE)	如果编译 init 时使用了 ALLOW_LOCAL_PROP_OVERRIDE 选项，并且 ro.debuggable 属性被设为了 1，那么就会加载这个文件。这使得开发者可以通过在 /data 分区里 push 一个文件的方式，修改之前的设置
/data/property/persist.* (PERSISTENT_PROPERTY_DIR)	重启后不会丢失的属性（Persistent property），这些属性文件会被加上前缀 persist。它们会被分别存放在 /data/property/ 目录中的各个文件里，躲过重启。只要 /init.rc 脚本中有 load_persist_props 这条指令，init 就会重新加载这些属性

另外还有一个属性文件 PROP\_PATH\_FACTORY(/factory/factory.prop)，以前 Android 中确实

1 这就是一个类似 Windows 注册表的东西，每个属性实际上也就是一个 key-value（键-值）对。——译者注



曾经有过这个宏定义，但现在已经不再支持了。请注意：各个属性文件加载的顺序是非常重要的，因为先后两次设置同一个属性的话，后一次的设置会覆盖掉前一次设置的结果——除非该属性被标记为“只读”。

因为 `init` 是系统中所有进程的祖先，所以只有它才天生适合实现系统属性的初始化。在它刚开始初始化的时候，`init` 中的代码会调用 `property_init()` 去安装系统属性。这个函数（最终）会调用 `map_prop_area()` 函数，并打开 `PROP_FILENAME`（这个宏定义指的是 `/dev/__properties__`），然后在关闭这个文件的描述符之前，用 `mmap(2)` 系统调用以“读/写”权限把这个文件的内容 `map` 到内存里。之后，`init` 又会再次打开这个文件，不过这次用的是 `O_RDONLY`，然后再 `unlink` 掉它。

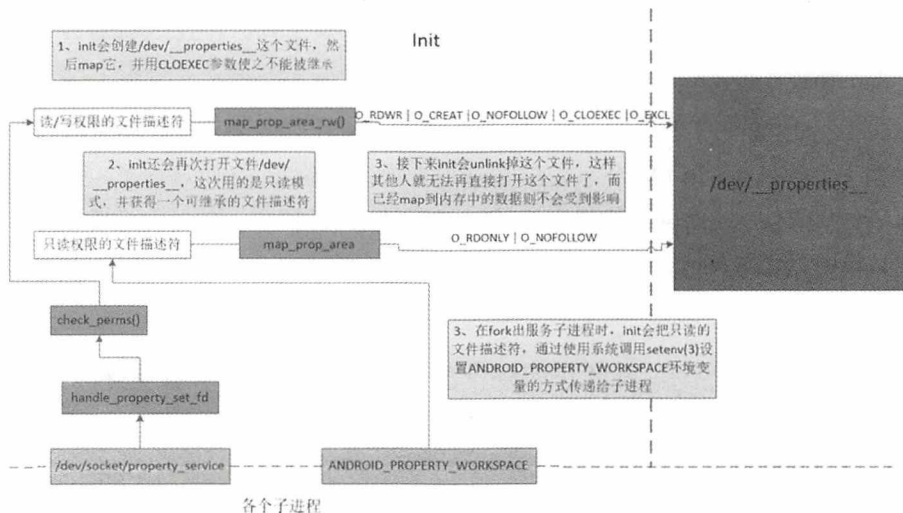


图 4-1 系统属性所在内存空间的 `map` 过程

属性文件的只读文件描述符被设为可以被子进程继承。这使得系统中任何一个进程都能方便地通过 `mmap(2)` 映射这个文件描述符的方式访问到系统的属性——尽管只能读。这是一个很巧妙的方法，它让所有可以访问属性的用户共享了这块被称为 `__system_property_area` 的物理内存<sup>1</sup>（它的大小是用宏定义 `PA_SIZE` 来表示的，这个宏定义的默认值是 128KB）。不过，对这块

1 `/dev/__properties__` 文件被 `mmap` 到内存里去之后，`map` 了该文件内容的内存区域的起始地址就会被记录到全局变量 `__system_property_area` 上，所以可以大致认为 `/dev/__properties__` 和 `__system_property_area` 指的是一个东西。事实上，如果你仔细些的话，甚至会发现并不真正存在 `/dev/__properties__` 这个文件——因为它是在 `/dev` 目录下的，没有哪个分区会被 `mount` 到这个目录上，在这个目录中都是一些伪文件，所以 `/dev/__properties__` 更像是 `__system_property_area` 在文件系统上的一个 `stub`。——译者注



\_\_system\_property\_area 内存区域进行写操作的权力，则被 init 独揽了。

你只要看一下 /proc 中各个进程目录里的 maps 伪文件的内容，就可以轻易验证：系统中所有用户模式下的进程都共享了这块内存区域，如输出结果 4-1 所示。

```
# In init: (note area is writable)
root@generic:/ # grep __properties /proc/1/maps
b6f2f000-b6f4f000 rw-s 00000000 00:0b 1369 /dev/__properties_
# In any user mode process (in this case, the shell)
root@generic:/ # grep __properties /proc/$$/maps
b6e5a000-b6e7a000 r--s 00000000 00:0b 1369 /dev/__properties_
```

输出结果 4-1 通过 /proc 伪文件系统，观察 \_\_system\_property\_area 的映射情况

大多数开发者对于 \_\_system\_property\_area 内部的结构都不是很清楚。这块内存区域的开头部分是个很短的头部，其中记录了一个序列号（表示内部版本号）、一个文件签名（0x504f5250 或者你也可以把它认作字符串“PROP”）以及一个版本号（对于较新版本的 Android 来说，它是 0xfc6ed0ab，如果出于兼容性的考虑，它是 0x45434f76）。紧接着这个头部的是 112 个字节的填充数据（为了把头部填满 128 个字节）。再接下来的就是系统的各个属性了。系统的各个属性被存储在一种混杂使用了单词查找树 [trie 树，亦称“前缀树”（prefix tree）或字典树] 和二叉树的数据结构中。这种数据结构在 Bionic 的 system\_properties.c 文件中有相当明晰的文档，如代码清单 4-1 所示。

代码清单 4-1 system\_properties.c 里记录的系统属性的内部结构

```
/*
 * Properties are stored in a hybrid trie/binary tree structure.
 * Each property's name is delimited at '.' characters, and the tokens are put
 * into a trie structure. Siblings at each level of the trie are stored in a
 * binary tree. For instance, "ro.secure"="1" could be stored as follows:
 */
*
* +-----+ children +-----+ children +-----+
* | | |----->| ro |----->| secure |
* +-----+
*
* left / \ right left / \ prop +-----+
* v v v v +----->| ro.secure |
* +-----+ +-----+ +-----+ +-----+
* | net | | sys | | com | | 1 |
* +-----+ +-----+ +-----+ +-----+
*/
```

## property\_service

为了能提供写（系统属性）请求的服务，init 专门打开了一个专用的 UNIX domain socket——/dev/socket/property\_service。只要能连上这个 socket，任何人都可以对它进行写操作（0666）。这些（通过/dev/socket/property\_service 这个 socket）写入的命令会被直接送给 init，init 首先会去检查 socket 的调用者（caller）有没有设置相关属性的权限——相关权限是写死在一个名为





“property\_perms”（且成员还在不断增长中的）数组中的。权限的检查方式是基于检查和比较 UID 和 GID [即，检查发起属性修改请求的用户的 UID 和 GID 与 “property\_perms” 中规定的 UID 和 GID 是否相符，init 可以从 property\_service 这个 socket 的调用者证书（caller credential）那里得到这些信息] 的。而系统写死在 property\_perms 数组中的 UID 和 GID，则如表 4-3 所示。UID 0 当然对所有属性拥有完全的访问权限。如果系统经过了 SELinux 增强（KitKat 和 L 及以后的版本），属性的 namespace 还会改用安全上下文（security context，定义在 /property\_contexts 中）提供更好的保护。系统属性的 namespace 和 UID 详见表 4-3（关于 SELinux 的讨论，详见第 8 章）。

表 4-3 属性的 namespace 及相关权限

namespace	拥有者的 UID	内 容
net.rmnet0 net.gprs net.ppp net.qmi net.ltr net.cdma	AID_RADIO	rild 使用的网络属性
gsm		GSM 相关设置
persist.radio		重启后仍然有效的 radio 设置
net.dns		域名解析模块（DNS resolver）设置（位于 /etc/resolv.conf）
sys.usb.config		USB 工作模式（adb、mtp、大容量存储设备、rndis 等）
net	AID_SYSTEM	所有的网络设置（包括那些拥有者是 AID_RADIO 的）
dev		所有硬件部件的设置
runtime		未使用
hw		硬件相关的设置
[persist.]sys		系统相关的设置
[persist.]service		启动/停止相应服务
persist.security		安全相关的设置
wlan		无线网络（Wi-Fi）设置
selinux		SELinux（安全增强的 Linux）设置
dhcp	AID_SYSTEM AID_SYSTEM AID_DHCP	DHCP 设置

续表

namespace	拥有者的 UID	内 容
debug	AID_SYSTEM AID_SYSTEM AID_SHELL	调试设置
log	AID_SHELL	日志设置
service.adb.root	AID_SHELL	供以 root 权限在运行的 ADB 使用
service.adb.tcp.port	AID_SHELL	供通过 TCP/IP 协议与电脑连接的 ADB 使用
sys.powerctl	AID_SHELL	电源管理控制
bluetooth	AID_BLUETOOTH	蓝牙设置
persist.service.dbroid		与 BlueDroid 协议栈相关的蓝牙设置

特殊的 namespace 前缀

init 能够识别出一些特定的前缀，这些前缀表示该如何处理相关属性。

- **persist 伪前缀**：设置这类属性是为了让它们系统在重启之后依然生效，这些重启后不会丢失的属性会被保存在/data/property/目录里的各个文件中，这些文件的拥有者（owner）必须是 root:root，且不能有链接。
- **ro 伪前缀**：这是用来标记只读（read-only）属性的。这些属性类似于 C 语言中的常量，不管操作者是不是它的拥有者，它能且只能被设置一次。通常，这些属性应该尽可能早地被设置，即在厂商提供 build 文件时就应该已经被设置好了。
- **ctl 前缀**：这是为了方便地控制 init 中提供的服务而设立的——通过把相关服务的服务名设为 ctl.start 或 ctl.stop 属性的值，就能很方便地启动或停止相关服务。（toolbox 中的 start 和 stop 工具实际上也就是靠这种带 ctl 前缀的属性来控制 zygote、surfaceflinger 和 netd 的）。在 control\_perms 数组中记录了一个不连续的 ACL（Access Control List，访问控制列表），通过 UID/GID 来限制谁有权启动或停止特定的服务。在 KitKat 中，这个访问控制列表添加了 dumpstate(shell:log)和 ril-daemon(radio:radio)这两项。从 Lollipop 版开始，SELinux 接管了 ACL 的访问控制职能。

属性的访问方法

在 toolbox 里提供了两个属性访问的命令行工具 getprop 和 setprop，另外还有一个属性监听工具 watchprops 命令。而与属性访问相关的原生 API 都定义在 system/core/include/cutils/property.h 文件中：

- int property\_get(const char \*key , char \*value, const char \*default\_value)——获取指定属性

键 (key) 的值 (value)。如果这个键的值不存在的话, 程序员也可以用参数 `default_value` 给它设定一个默认值。

- `int property_set(const char *key, const char *value)`——设定指定属性键的值。这个函数会把键和值串在一起, 通过 `property_service` 这个 socket, 发送给 `init`。
- `int property_list(void (*propfn)(const char *key, const char *value, char *cookie), void cookie)`——用一个回调函数——处理各个属性。只要有一个属性, 这个回调函数就会被调用一次, 调用时可以给这个回调函数传入一个预先指定的参数 `cookie`。

在 `<sys/_system_properties.h>` 文件中还定义了一些没有文档支持的 (但仍然能用的) 函数, 其中最有用的当数 `_system_property_wait_any(unsigned int serial)`, 它会让调用它的进程进入休眠状态, 直至某个属性被修改时才唤醒该进程。在 `watchprops` 命令中就用到了这个函数。

在框架中, 如果要访问属性的话, 需要使用 `android.os.SystemProperties`——这个类会通过 JNI 调用上述 API, 来访问系统属性。

## 实验: 使用 watchprops

`watchprops` 工具是用来实时监视系统属性变化情况的。尽管我们可以在设备启动过程中, 尽可能早地启动这个工具 (在电脑端上输入 `adb wait-for-device` 或 `adb shell watchprops` 命令), 但是 `build.prop` 及其之前加载属性的行为<sup>1</sup>还是不会被监视到的 (因为那个时候 `adb` 本身都还没有启动)。不过尽管是这样, 这仍然会揭示出一些在启动过程中设置的重要属性, 如带注释的代码清单 4-2 所示。

代码清单 4-2 系统启动过程中设置的系统属性, 已加注释

```
# Property (timestamp omitted)
persist.sys.dalvik.vm.lib.2 = 'libart.so'
sys.sysctl.extra_free_kbytes = '24300'
wlan.driver.status = 'unloaded'
net.hostname = 'android-35f4ac3424467a0f'
net.change = 'net.hostname'
persist.sys.profiler_ms = '0'
debug.force_rtl = '0'
net.qtaguid_enabled = '1'
net.change = 'net.qtaguid_enabled'
sys.settings_global.version = '1'
service.bootanim.exit = '1'
sys.boot_completed = '1'
dev.bootcomplete = '1'
init.svc.bootanim = 'stopped'
ro.runtime.firstboot = '1418626370520'
# gsm.* properties set by rilctl/libril
gsm.version.ril-impl = '...'
gsm.network.type = 'Unknown'
...
# rilctl also sets parameters, based on connection detected
sys.sysctl.tcp_def_init_rwnd = '60'
net.dns1 = '10.0.2.3'
```

Set by  
Zygote decides between Dalvik and ART  
ProcessList::updateOomLevels()  
ConnectivityService()  
internally, when a net.\* prop has changed  
SamplingProfiler's SettingObserver  
android.provider.Settings()  
SurfaceFlinger (causes bootanimation exit)  
ActivityManagerService::finishBooting()  
ibid, when not decrypting filesystem  
internally, once bootanimation has exit  
BootReceiver::logBootEvents()

1 见表 4-2, 属性文件的加载顺序。——原注



## .rc 文件

init 的主要操作是加载它的配置文件，并执行配置文件中的相关指令。传统上使用的配置文件有两个：主配置文件 `/init.rc` 和与设备相关的配置文件 `/init.hardware.rc`，其中斜体的 *hardware* 应该被替换为从内核参数 `androidboot.hardware`，或是从 `/proc/cpuinfo` 伪文件那里获取的字符串。比如说，模拟器中默认使用的这个 `hardware` 字段就是“goldfish”（在 M 版中是“ranchu”）。不过我们在实体机里，看到系统使用了 `/init.goldfish.rc` 文件的情况也不在少数，这可能就是因为大部分这类手机安装的 Android 系统直接复制了（模拟器）默认的文件系统，而没有注意到这些细节的缘故吧。这一设计的最初目的是：让所有的 Android 设备都能共享同一个 `/init.rc` 文件，而与设备相关的特定配置则让厂商写在 `/init.hardware.rc` 文件里。但是在实践中，我们也经常发现，许多开发者为了图省事，把一些指令直接加在了 `/init.rc` 文件中。

从 JB 版开始，唯一一个被硬编码写死的 rc 文件就是 `/init.rc` 了，其他 rc 文件可以用 `import` 指令显式地导入。JB 版默认的 `/init.rc` 也会导入 `/init.hardware.rc` 文件（以 `import /init.${ro.hardware}.rc` 的形式，其中 `${ro.hardware}` 表示相应属性的值）和 `/init.usb.rc`（或 `/init.${ro.hardware}.rc`）<sup>1</sup> 文件（这个文件中含有需要 init 执行的与 USB 设置相关的指令），详见本章下一节的讨论。另一个配置文件 `/init.trace.rc` 也出现在了默认的 build 版里，这是为了启用内核中用于调试的 `ftrace` 功能（详见第 2 本书中的讨论）。

### trigger、action 和 service

.rc 文件是由 `trigger` 语句块和 `service` 语句块构成的。`trigger` 语句块中的命令，会在触发条件被满足时执行；而 `service` 语句块中定义的是各个守护进程，init 会根据命令启动相关守护进程，也可以根据相关参数（“OPTION”类关键字）修改服务的状态。`service` 语句块由关键字 `service` 开头，后面跟着服务名及命令行。`trigger` 语句块由关键字 `on` 定义，后面跟着一个参数——这个参数既可以是预先规定的各个启动阶段（boot stage）的名称，也可以是一个 `property` 关键字，后面跟冒号加“属性名称=属性值”这样的表达式（在这种情况下，如果触发条件被满足，相应属性的属性值会改成指定的值）。在执行指定的动作（action）或命令（command）时，init 会分别把属性 `init.action` 或 `init.command` 的值设为当前正在执行的动作的名称或当前正在执行命令的名称。预先规定的启动阶段已经在表 4-4 中给出了，但是请注意，并不是每个启动阶段都必须使用，而且一些厂商还常常不遵守相关的约定（比如，在 `init` 阶段就去 `mount` 文件系统）。

---

1 原文如此，显然是复制-粘贴错了，这里应该是 `/init.${ro.hardware}.usb.rc`。——译者注

表 4-4 init 中各个启动阶段

启动阶段	内 容
early-init	初始化的第一个阶段，用于设置 SELinux 和 OOM
Init	创建文件系统，mount 点以及写内核变量
early-fs	文件系统准备被 mount 前需完成的工作
fs	专门用于加载各个分区
post-fs	在各个文件系统（/data 分区除外）mount 完毕之后需要执行的命令
post-fs-data	解密/data 分区（如果需要的话），并 mount 之
early-boot	在属性服务（property service）初始化之后，启动剩余内容之前需完成的作业
boot	正常的启动命令
charger	当手机处于充电模式时，需要执行的命令

init.rc 的语法和命令集

init.rc 及其导入的文件都有非常好的注释——只不过太长了些。与其不断地复制/粘贴它们，白白浪费纸张和读者您的感情，我们接下来，还是把讲解的重点放在这些脚本文件所使用的语法，以及其他一些相对而言没什么文档支持或者大家知之甚少的特性上。在阅读这一节内容时，你或许需要在手边准备一个/init.rc 文件，以便随时查阅参考。

init\_parser.c 中的代码在解析 rc 文件时，能够识别出两类关键字：COMMAND [也就是在 trigger 语句块或各个启动阶段中要执行的动作（action），这类关键字只在“on”关键字开头的语句块中有效] 和 OPTION（修改相关服务的状态，这类关键字只在“service”关键字开头的语句块中有效）。表 4-5 中列出了 init 所支持的各个 COMMAND 类的关键字（相关定义见“keywords.h”文件）。

表 4-5 init 支持的各个 COMMAND 类关键字

命 令	语 法
bootchart_init	M 版中引入：启用启动时的信任链验证（如果可以配置的话）。M 版中允许配置启动时的信任链验证
chdir <i>directory</i>	等价于 cd 命令（调用 chdir(2)）
chmod <i>octal_perms file</i>	修改指定文件的权限（以八进制数的形式表示文件的权限）
chown <i>user group file</i>	效果等价于 chown <i>user:group file</i> 命令
chroot <i>directory</i>	等价于 Linux 的 chroot 命令（调用 chroot(2)）
class_reset <i>service_class</i>	JB 版中引入：停止与 <i>service_class</i> 相关的所有服务

续表

命 令	语 法
<code>class [start stop] class</code>	启动或停止 <code>class</code> 参数指定的 <code>service_class</code> 相关的所有服务
<code>copy src_file dst_file</code>	类似 <code>cp(1)</code> 命令
<code>domainname domainname</code>	把指定的域名 ( <code>domainname</code> ) 写入 <code>/proc/sys/kernel/domainname</code> 伪文件
<code>exec command</code>	已经不再支持了
<code>enable service</code>	L 版中引入：启动一个已经被 <code>disable</code> 了的服务
<code>export variable value</code>	在全局环境中，设置环境变量 <code>variable</code> 的值。该命令的执行结果会影响到所有的进程
<code>hostname hostname</code>	把主机名（写入） <code>/proc/sys/kernel/hostname</code> 伪文件中
<code>ifup interface</code>	激活指定的网卡（ <code>interface</code> ）（类似于 <code>ifconfig interface up</code> 命令）
<code>insmod module.ko</code>	加载一个内核模块
<code>Import filename.rc</code>	添加另一个 <code>rc</code> 文件
<code>load_all_props</code>	L 版中引入：（重新）加载 <code>build</code> ， <code>default</code> 和 <code>factory</code> 文件中规定的属性
<code>load_persist_props</code>	JB 版中引入：（重新）加载 <code>/data/propert</code> 目录中的各个文件中的属性
<code>loglevel level</code>	设置内核的日志级别（ <code>loglevel</code> ）（ <code>printk</code> ）
<code>mkdir directory</code>	创建一个目录（调用 <code>mkdir(2)</code> ）
<code>mount fstype fs point</code>	把指定文件系统类型的（ <code>fstype</code> ）分区（ <code>fs</code> ） <code>mount</code> 到指定的 <code>mount</code> 点（ <code>point</code> ）上去
<code>mount_all</code>	<code>mount</code> 所有在 <code>vold</code> 守护进程使用的 <code>/fstab.hardware</code> 中规定的文件系统。这会使 <code>init</code> 进程 <code>fork()</code> 出一个子进程，并在这个子进程中用 <code>fs_mgr</code> 执行 <code>mount</code> 操作。 <code>Init</code> 同时也能检测出所有加密的文件系统
<code>powerctl shutdown/reboot</code>	KK 版中引入：是对 <code>shutdown/reboot</code> 命令的封装
<code>[re]start service_name</code>	启动/重启服务名与参数 <code>service_name</code> 相一致的 <code>service</code> 语句块中规定的服务
<code>restorecon[ -recursive] path</code>	用 <code>path</code> 参数指定的文件重新加载 SELinux 上下文（ <code>context</code> ）
<code>rm[dir] filename</code>	JB 版中引入：删除一个文件或一个目录（会分别调用 <code>unlink(2)</code> / <code>dmdir(2)</code> ）
<code>setcon SEcontext</code>	JB 版中引入：设置(或改变)SELinux 的上下文。 <code>init</code> 使用的上下文是 <code>u:r:init:s0</code>
<code>setenforce [0 1]</code>	JB 版中引入：强制启用或关闭 SELinux
<code>setkey table index value</code>	设置键盘映射表的内容
<code>setprop key value</code>	设置指定的系统属性
<code>setsebool name value</code>	设置某个与 SELinux 相关的布尔型属性。参数 <code>value</code> 可以是 <code>0/false/off</code> 或者 <code>1/true/on</code>



续表

命 令	语 法
<code>setrlimit category min max</code>	使用 <code>setrlimit(2)</code> 系统调用，设置进程可用资源的上下限（参见 <code>ulimit(1)</code> ）
<code>stop service_name</code>	停止服务名与参数 <code>service_name</code> 相一致的 <code>service</code> 语句块中规定的服务
<code>swapon all..</code>	KK 版中引入：激活所有 <code>fstab</code> 中的 <code>swap</code> 分区
<code>symlink target src</code>	创建一个符号链接（等价于 <code>ln -s</code> ，即调用 <code>symlink(2)</code> ）
<code>sysclktz tzoffset</code>	设置系统时区（使用 <code>settimeofday(2)</code> ）
<code>trigger trigger_name</code>	激活一个 <code>trigger</code> 语句块（这会使 <code>init</code> 重新运行该语句块中的相关命令）
<code>verity_load_state</code>	M 版中引入：加载 DM-verity 状态
<code>verity_update_state mount</code>	M 版中引入：更新 DM- Verity（分区加密状态），并设置系统属性 <code>partition.mount.verified</code> 的值
<code>wait file timeout</code>	等待文件 <code>file</code> 创建完毕，等待的最大时间不超过 <code>timeout</code> 秒
<code>write file value</code>	把 <code>value</code> 写到文件 <code>file</code> 中去。等价于执行 <code>echo value &gt; file</code> 命令

如果你把 `/init.rc` 文件通读一遍，可能就会发现这些命令（`COMMAND` 类的关键字指令）会被用在各个不同的启动阶段中，去执行一些我们可能需要在系统启动过程中执行的操作，比如设置目录结构、设定文件系统中的权限以及通过 `/proc` 或 `/sys` 设置一些内核参数。在写好了所有启动阶段中需要执行的指令之后，`rc` 文件中的余下部分就要用来写与服务相关的指令了。按照规定，`service` 语句块中使用的必须是用 `OPTION` 类的关键字表示的指令。这些指令的参数使 `init` 能够确定该如何运行相关服务，以及相关服务退出/崩溃时是否要执行某些操作。表 4-6 中列出了各个可用的 `OPTION` 类关键字。

表 4-6 init 支持的各个 OPTION 类关键字

OPTION 类关键字	说 明
<code>capability</code>	支持 Linux 的 <code>capabilities(7)</code> （或者至少在将来的某个时间会支持）
<code>class</code>	把服务加入某个服务组（ <code>service group</code> ）。可以用 <code>class_[start stop reset]</code> 命令同时操作组中的所有服务
<code>console</code>	把该服务定义为一个 <code>console</code> 服务， <code>stdin/stdout/stderr</code> 会被 link 到 <code>/dev/console</code> 上去
<code>critical</code>	把该服务定义为一个关键（ <code>critical</code> ）服务，关键服务（万一崩溃了的话）会自动重启。如果在 <code>CRITICAL_CRASH_WINDOW(240)</code> 秒之内，它的崩溃次数超过了 <code>CRITICAL_CRASH_THRESHOLD(4)</code> 次，系统将会自动重启到 <code>recovery</code> 模式
<code>disable</code>	表示该服务不需要启动，但该服务之后还是可以手工启动的

续表

Option 类关键字	说 明
group	规定该服务应该以指定的 gid 启动。init 会调用 setgid(2)来完成这一操作
ioprio	指定该服务的 I/O 优先级，init 会调用 ioprio_set 来完成这一任务
keycodes	指定触发该服务的组合键（key chord，详见下一节讨论）
oneshot	告诉 init 启动该服务，然后就不用管它了（即，忽略掉 SIGCHLD 信号）
onrestart	列出该服务重启时要执行的命令，通常用来重启其他依赖服务（dependent service）
seclabel	指定应用在该服务上的 SELinux 标签（label）
setenv	在该服务被 fork()出来并被 exec()之前，先给它设置一个环境变量。不像表 4-5 中的 export 关键字，这个环境变量只有该服务才能看见
socket	告诉 init 打开一个 UNIX Domain socket，并让该服务进程继承这个 socket，这样做是为了解决服务的 stdin/sdout 的设置问题——让服务不必担心应该打开哪个 socket 以及它是不是有权限使用这个 socket
user	指定该服务应以 uid 身份运行，init 将调用 setuid(2)完成这一任务
writepid	M 版中引入：把子进程的 PID 写到指定的文件中，用于设置 cgroups 资源控制

启动服务

尽管语法不尽相同，但是 Android 中的 init 还是用 pid=1 进程（也就是 init 和 launchd）的传统方式启动服务的，即它 fork()出服务子进程，然后（调用 setuid(2)/setgid(2)）设置服务子进程的权限，并设置该服务子进程用来获取输入的（UNIX Domain）socket 及配置环境变量、I/O 优先级（在服务语句块中使用 ioprio 关键字）和 SELinux 上下文。对于用关键字 console 定义的服务，init 会把/dev/console 连到它的 stdin/stdout/stderr 上，而对于其他服务，它会把 stdio 给“干掉”。尽管当前还不支持，但是在将来，init 应该也会去设置用关键字 capability 定义过的服务的权能集（capability set）（详见第 8 章的讨论）。只有当所有这些操作都执行完毕之后，init 才会去执行服务本身的二进制可执行文件。

在服务启动之后，init 会维持一个指向该服务的父链接（parental link）——这样，一旦该服务停止运行或者崩溃了，init 就会收到一个 SIGCHLD 信号，并注意到这一事件，然后重启该服务。onrestart 关键字会使 init 在各个指定的服务之间建立起关联，当特定的服务需要重启时，init 会去运行这个使用了 onrestart 关键字的 service 语句块中的命令，或者重启与该服务有依赖关系的服务。critical 关键字定义的是系统“必需的”服务。如果 init 发现某个用 critical 关键字定义过的服务不断地重启（也就是 init 重启该服务，只会让它再次崩溃），它就会把整个系统重启到 recovery 模式下。对于每一个服务，init 还会维持一个反映了该服务当前运行状态

(running/stopped/restarting) 的属性 `init.svc.service`。

## 组合键 (keychords)

`init` 有个很有意思 (但大家却知之甚少) 的功能——`init` 可以在用户按下某个组合键 (keychords) 时启动某些服务。“组合键”被定义为用户在任意一个时刻同时按下某些键 (对于那些带有物理键盘的设备) 或某些按钮的行为 (有点像弹钢琴时同时按下几个琴键那样)。每个键都有指定的对应 ID——它们是从 Linux 的 `evdev` 输入机制那里得到的。

需要注意的是: 写在 `keycodes` 关键字后面的, 应该是 Android 键盘布局文件 (key layout file, 通常放在 `/system/usr/keylayout` 目录里) 中规定的键位代码 (code), 而不是 Android 框架规定和使用的键位代码 (即 `frameworks/native/include/android/keycodes.h` 中规定的键位代码)。唯一一个绑定了组合键的默认系统服务是 `bugreport`, 在有些设备 (比如 Nexus 5) 上, 只要同时按下音量键和电源键就能启动该服务。你可以在 Nexus 手机上的 `/init.hammerhead.rc` 文件中找到相关定义, 如代码清单 4-3 所示。

代码清单 4-3 `/init.hammerhead.rc` 中能证明 BugReport 服务使用了组合键的代码

```
service bugreport /system/bin/dumpstate -d -p -B \
    -o /data/data/com.android.shell/files/bugreports/bugreport
class main
disabled
oneshot
keycodes 114 115 116
```

`dumpstate` 命令是 AOSP 中提供的一个二进制可执行文件, 它会逐一列举所有的子系统和服 务, 给出它们的所有诊断和实时使用信息。注意, `bugreport` 正常情况下是被禁用 (disabled) 的, 这也就是说, 它需要手工启动, 而它绑定的组合键的键位代码是 114、115 和 116。查询一下 `/system/usr/keylayout/Generic.kl` 中的相关定义, 你就会发现, 它们各自对应的是 `VOLUME_DOWN`、`VOLUME_UP` 和 `POWER`。

要支持组合键, `/dev/keychord` 文件是必须存在的。这个文件是一个由 `keychord` 内核驱动导出的设备节点。如果内核编译时使用了 `INPUT_KEYCHORD`, 或者该驱动已经安装为一个模块了, 这个驱动就可以被认为具有某些 “Android 的意味”, 我们将在第 2 本中详细讨论它。



在一个已经 root 了的设备 (也就是 root 文件系统被修改过的设备) 上, 你可以用 “组合键” 给它添加各种各样的功能。在默认配置中, 你还受到某些限制 (因为你只能用实体按钮构造各种组合键), 但你还是可以添加各种按钮组合去实现各种功能——只要你想得出来。而在那些自带了物理键盘或者额外的其他按键的设备上, 用组合键能玩的花样就更多了。



## mount 文件系统

尽管 Android 有一个专门的卷管理守护进程（`vold`），`init` 仍需亲自执行一些 `mount` 操作。我们回忆一下，当 `init` 进程刚启动时，只有 `root` 文件系统被 `mount` 了上来，此时既没有 `/system` 也没有 `/data`，所以 `init` 就面临这样一个窘境：它至少得把 `/system` 给 `mount` 上来吧，只有这样，（`/system` 上的）各个守护进程（也包括 `vold` 在内）才能启动。显然，这是个很关键的操作：如果无法把文件系统 `mount` 上来，`/init` 会把系统重启到 `recovery` 模式下。

`init` 能够认出 `/init.rc` 中的 `mount_all` 指令（通常位于 `on fs` 语句块中），并执行 `mount` 所有默认的文件系统的操作。“默认的文件系统”是在 `/fstab.hardware` 文件（这是 AOSP build 的文件之一）中规定的。执行 `mount` 操作的代码位于 `fs_mgr` 中，无论 `init` 还是 `vold` 都会使用它。当 `init` 执行 `mount` 操作时，它首先会 `fork()` 出一个子进程执行相关操作——这是为了避免因为在 `mount` 的过程中出现问题，而影响到 `init` 自己的启动活动。

被 `fork()` 出来的子进程会去执行 `mount` 操作，如果需要的话，还会对文件系统进行 `fsck` 操作。`fs_mgr` 中规定了对各种不同的文件系统执行 `fsck` 操作的程序所在的路径（当前是 `/system/bin/e2fsck`，而在 L 及以后的版本中，则是 `/system/bin/fsck.f2fs`），而且这些 `fsck` 操作也是由再次 `fork()` 出来的子进程完成的。`fs_mgr` 中的代码会提升它的日志级别（`logging level`），所以你可以在内核 `ring buffer` 中看到 `fs_mgr` 输出的各种消息（使用 `dmesg`）。如果你在足够早的时候，使用 `dmesg` 获取到了内核 `ring buffer` 中的消息（只要在新写入 `ring buffer` 中的消息还没有来得及覆盖掉 `fs_mgr` 输出的消息之前就行了），你就会看到这些带有 `EXT4-fs` 和 `F2FS-fs`（这两个内核模块是用来支持对应文件系统的）以及 `SELinux`（如果检测到文件系统使用了相应的扩展属性时，会强制应用 `SELinux`）这类 `fs_mgr` 特有标记的消息。

很显然，`fork()` 出来执行 `mount` 操作的子进程会（和其他所有子进程一样）向父进程返回一个返回码。`/init` 会根据这个返回码设置属性 `vold.decrypt` 的值，在需要的时候，这个属性的值会被 `vold` 守护进程读取，并对相应的文件系统做解密操作。如果没有文件系统被加密，`init` 会去执行名为“`nonencrypted`”的 `trigger` 语句块中的指令。

## 总结：init 的执行流程

作为大多数守护进程的样板，`init` 代码的指令流程完全遵循建立服务的经典模式：初始化，然后陷入一个循环中，而且永远不想从中退出来。

## 初始化流程

init 进程的初始化工作由以下步骤组成：

- 检查自身这个二进制可执行文件是不是被当成 `ueventd` 或者（KitKat 及之后版本中的）`watchdogd` 调用的。如果是的话，余下的执行流程就会转到相应的守护进程的主循环那里去，我们将在本章稍后讨论这一部分内容。
- 创建 `/dev`、`/proc` 和 `/sys` 等目录，并且 `mount` 它们。
- 添加文件 `/dev/.booting`（通过打开这个文件，然后再关闭它的方式），在启动完毕之后，这个文件会被（`check_startup`）清空（见图 4-2）。
- 调用 `open_devnull_stdio()` 函数完成“守护进程化”操作（把 `/stdin/stdout/stderr` 链接到 `/dev/null` 上去）。
- 调用 `klog_init()` 函数创建 `/dev/__kmsg__`（Major 1，Minor 11），然后立即删除它。
- 调用 `property_init()` 函数，在内存中创建 `__system_property_area` 区域，相关讨论见本章之前的“系统属性”一节。
- 调用 `get_hardware_name()` 函数，读取 `/proc/cpuinfo` 伪文件中的内容，并提取出“Hardware”一行的内容作为硬件名（hardware name）。以这种方式获取硬件名，法子虽然糙了点，但确实有效（至少在使用 ARM 体系结构处理器的设备上是可以正常工作的）。
- 调用 `process_kernel_cmdline()` 函数，读取 `/proc/cmdline` 伪文件中的内容，并把所有 `androidboot.XXX` 的属性都复制一份出来，变成 `ro.boot.XXX`。
- 在 JellyBean 及以后的版本中，这里要初始化 SELinux。在 JellyBean 版本中，如果没有用 `#ifdef` 定义 `HAVE_SELINUX` 还不会启用 SELinux，而到了 KitKat 版，SELinux 就是默认启用的了。SELinux 的 context 是放在 `/dev` 和 `/sys` 里的。
- 接着还要专门检查一下设备是否处于“充电模式”（根据一个名为“`androidboot`”的内核参数进行判断）。如果设备处于“充电模式”的话，会使 `init` 跳过大部分初始化阶段，并且只加载各个服务中的“`charger`”类（当前也只有“`charger`”守护进程有这个类）。如果设备并没有处于“充电模式”，那么 `init` 将会去加载 `/default.prop`，正常执行启动过程。
- 调用 `init_parse_config_file()` 函数去解析 `/init.rc` 脚本文件。
- `init` 会把 `init.rc` 文件中各个 `on` 语句块里规定的 action（用 `action_for_each_trigger()` 函数）以及内置的 action（用 `queue_builtin_action()` 函数）添加到一个名为 `action_queue` 的队列里去。最终得到的队列如图 4-2 所示。

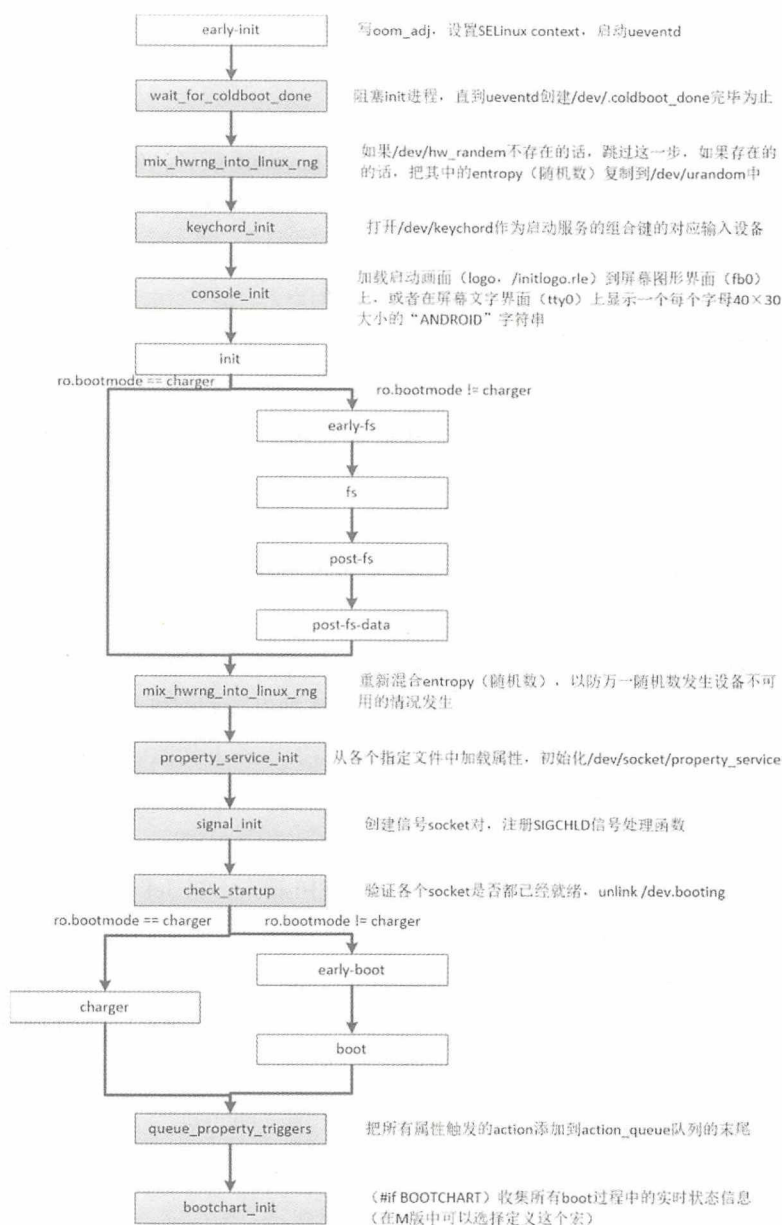


图 4-2 init 的启动阶段 (白色框) 和内置的命令 (灰色框)

最后, 主循环中将会逐个执行 init.rc 中的所有命令, 然后, init 将在它的生命周期中的大多数时间里处于休眠状态, 间或轮询一下各个文件描述符, 并根据宏 BOOTCHART 有没有被定义过, 决定是否将日志发送给 bootchart, 只有在必要的时候 init 进程才会被唤醒。我们可以通过



查看/proc 伪文件系统，来了解 init 进程打开了哪些文件描述符，如输出结果 4-2 所示。

```
root@generic:/proc/1 # ls -l fd
lrwx----- .... 0 -> /dev/_null_ (deleted) #
lrwx----- .... 1 -> /dev/_null_ (deleted) # stdin, stdout and stderr closed
lrwx----- .... 2 -> /dev/_null_ (deleted) #

l-wx----- .... 3 -> /dev/_kmsg_ (deleted) # Major: 1, Minor: 11

lr-x----- .... 4 -> /dev/_properties_ # read-only property store, for children

lrwx----- .... 5 -> socket:[1643] # property_set_fd (/dev/socket/property_service)

lrwx----- .... 6 -> socket:[1645] # signal_fd (socketpair[0])
lrwx----- .... 7 -> socket:[1646] # signal_rcv_fd (socketpair[1])

lrwx----- .... 9 -> socket:[1784]
```

输出结果 4-2 通过/proc/1/fd 查看 init 打开的文件描述符

## 主循环 (run-loop)

init 的主循环相当简洁，一共由三个步骤组成：

- `execute_one_command()`——从队列 `action_queue` 的头部取出一个 `action`（如果有的话），并执行之。
- `restart_processes()`——逐个检查所有已经注册过的服务，并在必要时重启之。
- 安装并轮询（监视）如下三个 `socket` 描述符。
  - `property_set_fd` (`/dev/socket/property_service`)，这个 `socket` 是用来让想要设置某个属性的客户端进程，通过它把要设置的属性的键(key，也就是属性的名称)和值(value)发送给 `init` 的。`property_service` 的代码会(使用 `getsockopt(..SO_PEERCREDS..)` 函数)获取 `socket` 对端进程的证书，并检查它是不是有权限来设置该属性。如果有权限的话，除了设置属性的值之外，因这个属性改变而会被触发执行的语句或者对应服务的状态（比如要设置的是 `ctl.start/stop` 属性）也会被执行或发生改变。如果启用了 SELinux，`/init` 还会调用它，强制检查 `/property_contexts`。
  - `keychord_fd` (`/dev/keychord`，如果存在的话），它是用来处理上文讨论过的启动服务的组合键的。
  - `signal_rcv_fd`，它是一对 `socketpair` 中的一端，创建它是为了处理因子进程死亡而来的 `SIGCHLD` 信号。当 `init` 收到这个信号之后，`sigchld_handler` 就会向 `socketpair(signal_fd)` 写入数据，这样在这个 `socketpair` 的接收端就能收到这些数据，并导致 `init` 去调用 `wait_for_one_process(0)`。这样就能获得已死亡的子进程的返回码（以便释放该进程残留的资源，防止它成为僵尸进程）。然后 `init` 会清理这个 `socketpair`

中的数据，等待下一个子进程挂掉。当然如果这个挂掉的子进程是个关键（critical）服务的守护进程的话，init 也会重启这个进程。

必须要强调的一个重点是：除了它监听的这三个文件描述符之外，/init 是不会从其他地方接收输入的。换言之，除此之外再也没有其他方法可以影响/init 的操作行为了。之所以做出这样的设计，是因为/init 是一个没有任何限制的以 root 身份运行的进程。唯一能够修改/init 的操作行为的方法就是编辑/init.rc 文件的内容，但这个文件是 root 文件系统的一部分，位于一个完全隔离的分区里（和内核放在一起），并有数字签名的保护，其中的内容几乎是不可能被修改的，除非是在一个已经解锁了 Boot Loader 的设备上。

## 4.2 init 和 USB

Android 设备经常会根据用户的选择，修改自己通过 USB 线连上电脑后，该以哪种设备的面目出现：到底是作为一个大容量存储设备，还是模拟一台数码相机；是不是要启用/关闭 ADB 调试功能，等等。Android 设备中没有一个专门的守护进程来处理这些问题，所以这一重任就落在了 init 的肩上——由它来和内核中的 USB 组件通信。

Android 设备用 USB 连上电脑后表现为哪种设备是由系统属性 sys.usb.config 决定的。各个框架（特别是 UsbDeviceManager 及其相关类）会根据用户的选择去设置这个属性的值。而 init 作为所有系统属性的管理者会在收到修改请求之后，修改 sys.usb.config 的值，然后根据改后的 sys.usb.config 值，执行对应 trigger 语句块中的指令。为了便于管理，所有这些与 sys.usb.config 的值相关的 trigger 语句块都被单独放在了 init.hardware.usb.rc 脚本文件中。这一点我们可以从下面这个实例中得到验证，这是从一台 Nexus 5 手机里提取出来的 init.hammerhead.usb.rc 文件内容（见代码清单 4-4）。

如代码清单 4-4 这段代码所示，init 除负责修改 sys.usb.config 属性的值之外，还要向 /sys/class/android\_usb/android0 目录中的各个伪文件中写入数据。这些伪文件的另一端（数据接收端）是 USB gadget driver。顾名思义，这是个多功能的驱动程序，它可以根据用户态进程发来的要求，把 Android 模拟成各种可以在电脑端上使用的 USB 设备。这个驱动是根据 sys.usb.config 属性的当前值，来决定自己的行为的。属性 sys.usb.config 可以使用的值，以及对应的模拟行为如表 4-7 所示。

代码清单 4-4 从一台 Nexus 5 手机里提取出来的 init.hammerhead.usb.rc 文件中与 USB 相关的设置代码

```
on init
    write /sys/class/android_usb/android0/f_rndis/manufacturer LGE
    write /sys/class/android_usb/android0/f_rndis/vendorID 18D1
    write /sys/class/android_usb/android0/f_rndis/wceis 1

on boot
    write /sys/class/android_usb/android0/iSerial $ro.serialno
    write /sys/class/android_usb/android0/iManufacturer $ro.product.manufacturer
    write /sys/class/android_usb/android0/iProduct $ro.product.model

# MTP
on property:sys.usb.config=mtp
    stop adb
    write /sys/class/android_usb/android0/enable 0
    write /sys/class/android_usb/android0/idVendor 18D1
    write /sys/class/android_usb/android0/idProduct 42E1
    write /sys/class/android_usb/android0/bDeviceClass 0
    write /sys/class/android_usb/android0/bDeviceSubClass 0
    write /sys/class/android_usb/android0/bDeviceProtocol 0
    write /sys/class/android_usb/android0/functions mtp
    write /sys/class/android_usb/android0/enable 1
    setprop sys.usb.state ${sys.usb.config}
..
```

表 4-7 USB gadget driver 能够识别的 sys.usb.config 属性的值

accessory	Android 设备作为 USB 从机与其他设备相连——需要实现 AoA 协议
acm	acm 是 Abstract Control Model（抽象控制模型）的缩写（USB modem）
adb	启用 Android 调试桥（adb）功能，创建/dev/android_adb 设备节点，电脑端可以通过这个设备节点与 Android 设备中的 adb 守护进程进行通信
audio_source	（当与外部扬声器连接时）把 Android 设备用作 USB 音源设备，并提供 PCM playback（放音）功能
mass_storage	把 Android 设备用作大容量存储设备（移动硬盘）
mtp	mtp 是 Media Transfer Protocol（多媒体传输协议）的缩写。在这种情况下，Android 设备会被认作一台数码相机，并且 Android 设备会在内核中创建一个线程，专门用来处理 MTP 请求，并创建/dev/mtp_usb 设备节点
rndis	也就是 USB Remote NDIS，实际上就是手机的 USB 共享上网的功能。即电脑可以通过数据线连接手机，然后手机 USB 共享网络，这样电脑就可以使用手机连接的 Wi-Fi 上网了

为了使设置生效，USB 驱动要被禁用然后再启用一下，这也就是为什么在启用“USB 调试”功能或者绑定手机 USB 共享上网功能时，从电脑端看，手机会先断开一下，然后再连上来（你可以观察到电脑端相关的内核消息，或者如果你用的是一台虚拟机的话，会看到一个弹窗）。

实验：修改 Android 设备的 USB 名称

下面这个实验向你演示了如何为你自己的 Android 设备定制一个个性化的名称。虽然这个实验是在一台 Galaxy S3 手机上完成的，但这个实验的做法在所有 Android 设备上都是行得通的（见输出结果 4-3）。



```

shell$3:/sys/class/android_usb/android0# ls
bDeviceClass
bDeviceProtocol
bDeviceSubClass
bcdDevice
enable                # Toggles enable/disable

f_accessory#
f_acm#
f_adb#
f_ccid#
f_diag#
f_mass_storage#      # Exported gadget
f_mtp#                # driver functions
f_ncm#
f_ptp#
f_rnnet#
f_rnnet_sdio#
f_rnnet_smd#
f_rnnet_smd_sdio#
f_rndis#

functions             # controls functionality
host_state
iManufacturer          # Holds vendor string (e.g SAMSUNG)
iProduct               # Holds product ID reported: e.g. SAMSUNG_Android_SGH-I747
iSerial                # Holds serial # reported by adb
idProduct              # vendor's product id
idVendor                # well known vendor id (e.g. Intel: 8086)
power
remote_wakeup
state
subsystem
terminal_version
uevent
shell$3:/sys/class/android_usb/android0 $ cat functions
mtp,acm,adb
shell$3:/sys/class/android_usb/android0 $ cat iProduct
SAMSUNG_Android_SGH-I747

```

输出结果 4-3 Galaxy S3 手机上记录 USB 标识信息的一些文件

修改 iProduct 文件中的内容就能改变这台手机在电脑端上显示的名称（比如，Kindle HDX 设备的默认名称是“Lab126 Android”）。修改 iSerial 文件的内容则会改变“adb devices”命令输出的设备串号（再接下来可以用“adb -s 设备串号”的方式选择相关设备），而 iManufacturer 和 iProduct<sup>1</sup>文件中的内容也可以用类似的方式予以修改。你可以在自己的手机上试一下，往相应的文件中写入你自己觉得合适的字符串，然后拔掉手机和电脑间的 USB 连接线后，再把它接上，看看相关的设备标识有没有真的改变。请注意：用这种修改方法做出的修改结果，在手机重启之后就没有了。不过如果你真想要给你的手机起个够酷的名字，其实也很方便，只要把它们写到你的手机上的/init.hardware.usb.rc 文件（或者手机上实现类似功能的文件）里去就可以了。

## 4.3 init 的其他角色

在本章的最后一节中，我们要讨论一下 init 承担的其他两个角色——那就是 ueventd 和（KitKat 及以后版本中的）watchdogd。虽然完成这些任务所使用的二进制可执行文件和/init 进程的二进制可执行文件是同一个文件，但是代码的执行路径却是完全不同的，而且具体怎样执

1 原文如此，但 iProduct 文件上面刚提到过，这里显然是 idProduct 之误。——译者注

行也取决于之前的一些初始化步骤的执行结果。

ueventd

在作为 ueventd 运行时，init 这个二进制可执行文件是用来管理硬件设备的：它需要响应内核通知，管理/sys 伪文件系统中与各个设备对应的文件并负责让各个进程能够通过它创建在/dev 中的符号链接访问到这些文件。完成这些操作时，它使用的是另一些初始化脚本文件——/ueventrd.rc 和/ueventrd.hardware.rc（其中斜体的 hardware 应该被替换为从/porc/cpuinfo 伪文件那里获取到的硬件名，或是从内核参数 androidboot.hardware 那里得到的字符串）。不过，和 init 进程所使用的 rc 脚本不同，这些配置文件里只记录了相关文件的路径及其应被设为什么样的权限。ueventd 会逐行处理这些 rc 文件里的每条记录，并调用 set\_device\_permission()函数，设置对应文件的权限（见图 4-3）。

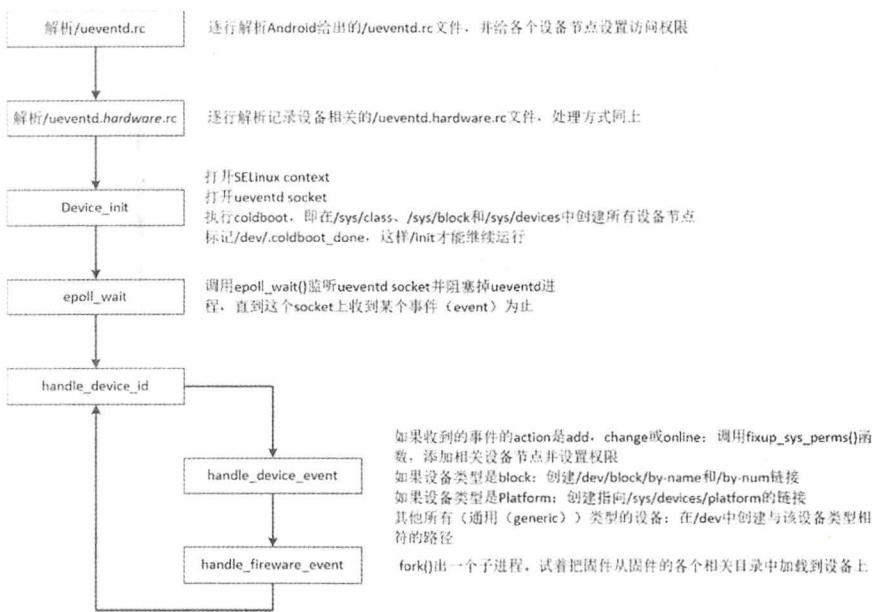


图 4-3 ueventd 的运行流程

接下来，ueventd 会去调用 device\_init() 函数，这个函数会去初始化一个名为“NETLINK\_UEVENT”的 socket。如果它发现设备还在 coldboot 过程中（即还没有创建 /dev/coldboot\_done 文件），ueventd 会分别在/sys/class、/sys/block 和/sys/devices 这三个子目录中添加一个名为 uevent 的伪文件。这会让那些在 uventd 启动前已经 add 上的 uevent 设备，重新再发一次通知消息（notification）。

一旦 (device\_fd) socket 初始化位完毕之后, ueventd 的任务就简单多了: 它只要不断地去轮询, 读取已经产生了的事件并处理它们就可以了。ueventd 处理的事件大致可以分成如下两大类。

- **设备事件 (device event):** 这些事件是在添加或删除设备时, 由各个内核子系统生成的。在收到这类事件时, ueventd 的功能与传统 Linux 的功能十分类似, 它会去创建或删除/dev 中与该设备相关的节点。
- **固件事件 (firmware event):** ueventd 需要监听 “firmware” (固件) 类设备的 “add” 事件, 如果收到了这类事件, 它会使用从/etc/firmware、/vendor/firmware 和/firmware/image 目录中加载固件的升级包。

如果编译时使用了-DLOG\_UEVENTS 参数, ueventd 还会把相关事件写在 INFO 消息中记入日志。

## watchdogd

和 ueventd 一样, watchdogd 也是 init 的另一面。这时, 它被用作硬件 watchdog 定时器(timer) (/dev/watchdog, 如果有的话) 在用户态中的接口。设置一个超时 (timeout) 变量, 每隔一段时间就发送一个 keepalive 信号 (一个 NULL 字节)。如果超过了超时变量规定的时间, watchdogd 还未能及时发送 keepalive 信号, 硬件 watchdog 定时器就会发出一个中断, 要求内核重启。尽管有那么点极端的感受, 但我们要知道唯一能让 watchdogd 不能及时发送 keepalive 信号的原因只有系统已经挂机 (hang) 了——这时系统很可能已经无法从挂机的状态恢复过来了, 重启设备反而算是个比较简单的解决方案。

作为 watchdogd 时, 这个守护进程只接受两个命令行参数: interval 和 margin——这两个参数的初始值都是 10, 单位是 “秒”。设备的总超时时间 (timeout 变量的值) 是这两个参数之和 (即, 默认值为 20 秒)。不过在超时没有接收到信息后, 系统并不会马上重启, 而是会再稍微等上一小会 (给系统最后一次补救的机会)。

## 本章小结

本章探讨了/init 的方方面面, /init 是最重要的系统进程, 没有它就没有用户态的一切。一开始我们把它和 Linux 与 UNIX 中的 init 进程做了一个比较。然后, 我们转而去探讨它最重要的独有特性——系统属性。接着, 我们又讨论了 rc 文件使用的语法, 并勾勒出了 init 运行的整个流程。

在下一章里, 我们将去探讨系统中的各个服务。我们会重点讨论各个默认的系统守护进程, 然后再去探讨为 Android 中所有框架提供支持的 system\_server。而那些实际上是由框架提供的服务 (它们的数量实在是太多了), 必须要从程序员的角度出发才能讨论更多的细节。这也就是



为什么我把它们留在第 2 本书中再予以讨论的原因。

本章讨论所涉及的文件

用作什么进程	文件/目录	其中的内容
init	system/core/init	/init 的源代码
	/system/core/init/readme.txt	相关命令和触发语句的文档
ueventd	system/core/init/ueventd.[c h]	init 被用作 ueventd 时的源代码
watchdogd	system/core/init/watchdogd.[c h]	init 被用作 watchdogd 时的源代码

# Android 的守护进程

Android 在后台运行着好几个守护进程 (daemon) 用来完成各种杂七杂八的事务性和操作性工作。这些服务的定义大多数散落在 `/init.rc` 的各个 `service` 语句块中，它们在 `/init.rc` 里出现的顺序并不重要，唯一影响它们启动顺序的是它们所属的分类。被分在“core”类中的服务会首先启动，接着启动的是被分在“main”类中的服务。`rc` 文件中也可以定义一个名为“late\_start”的服务类，供那些依赖于 `/data` 分区中的数据的服务使用，不过默认的服务里没有一个是属于这一类的。在这一节中，我们将使用这一服务分类方法——只是，因为大多数服务都属于“main”大类，所以我们还将把这些服务按照其功能进一步细分成各个子类。

紧接着上一章中我们对 `init` 进程的讨论，将讨论“core”类的服务 `adbd`、`servicemanager`，KitKat 中引入的 `healthd` 以及在 L 版中新增的 core 类服务 `lmkd` 和 `logd`。

其他所有的服务通常都被归在“main”大类下。所以接下来，我们又会把它们细分为网络类服务 (Network Services，这类服务包括 `netd`、`mdnsd`、`mtpd` 和 `rild`) 和图形及媒体类服务 (Graphics and Media Services，这类服务包括 `surfaceflinger`、`bootanimation`、`mediaserver` 和 `drmserver`)，剩下的服务很难归类，所以我们只好把它们全都放在“其他服务”这一分类下，这些服务包括 `installd`、`keystore`、`debuggerd`、`sdcard`，以及最后一个服务（但并非不重要）`Zygote`。

## 5.1 core 类中的服务

在用户模式启动过程中，首先运行的是被归在“core”类中的服务，这些服务都不需要访问 `/data` 分区，所以无论这个分区有没有被 `mount` 上来，都不会影响这些服务的运行。

### adbd

如果你是从头一路读到这儿的，我想 ADB 就不用我多介绍了吧：电脑和移动设备就是通

过这一媒介（也就是众所周知的 Android 调试桥<sup>1</sup>）进行通信的。我们既可以通过 adb 命令直接使用 Android 调试桥，也可以通过 ddms 命令间接地使用 Android 调试桥。adb 本身就自带了一篇非常好的文档：不输入任何参数，直接运行它，就会在屏幕上输出（相当详细的）用法说明。在我们的讨论中，我们更关心的是：ADB 到底是如何工作的。

在 Android 的默认配置下，adbd 作为提供 ADB 服务功能的设备守护进程，是定义在/init.rc 中的 [尽管它同时也被设为 disable（禁用）]（见代码清单 5-1）。当系统属性 sys.usb.config 的值中含有字符串“adb”时 [这是由地球人都知道的“USB 调试”（USB Debugging）图形界面选项激活的]，可以由/init.usb.rc 中的相关指令启动这一服务。

代码清单 5-1 adb 在 rc 文件中的定义（KitKat）

```
# adbd is controlled via property triggers in init.usb.rc
service adbd /sbin/adbd
    class core
    socket adbd stream 660 system system
    disabled
    seclabel u:r:adbd:s0 # As of JB, ADB gets its own SELinux context
```

注意，在默认情况下，adbd 是以 uid root 的权限启动的。不过它确实还会主动把自己降到 uid shell:shell，以及另几个组（group）的权限，如代码清单 5-2 给出的 adb.c 中的代码所示。

代码清单 5-2 adb\_main 启动函数中的权限设置的相关代码

```
int adb_main(int is_daemon, int server_port)
..

property_get("ro.adb.secure", value, "0");
auth_enabled = !strcmp(value, "1");

if (auth_enabled)
    adb_auth_init();

...

/* add extra groups:
** AID_ADB to access the USB driver
** AID_LOG to read system logs (adb logcat)
** AID_INPUT to diagnose input issues (getevent)
** AID_INET to diagnose network issues (netcfg, ping)
** AID_GRAPHICS to access the frame buffer
** AID_NET_BT and AID_NET_BT_ADMIN to diagnose bluetooth (hcidump)
** AID_SDCARD_R to allow reading from the SD card
** AID_SDCARD_RW to allow writing to the SD card
** AID_MOUNT to allow unmounting the SD card before rebooting
** AID_NET_BW_STATS to read out qtaguid statistics
*/

gid_t groups[] = { AID_ADB, AID_LOG, AID_INPUT, AID_INET, AID_GRAPHICS,
    AID_NET_BT, AID_NET_BT_ADMIN, AID_SDCARD_R, AID_SDCARD_RW,
    AID_MOUNT, AID_NET_BW_STATS };
```

<sup>1</sup> 有意思的是：三星的 Tizen 中使用的 sdb（smart debugger bridge）根本就是 ADB 的翻版，甚至连命令行的语法也完全一模一样。——原注



```

    if (setgroups(sizeof(groups)/sizeof(groups[0]), groups) != 0) {
        exit(1);
    }

    /* don't listen on a port (default 5037) if running in secure mode */
    /* don't run as root if we are running in secure mode */
    if (should_drop_privileges()) {
        drop_capabilities_bounding_set_if_needed();

        /* then switch user and group to "shell" */
        if (setgid(AID_SHELL) != 0) {
            exit(1);
        }
        if (setuid(AID_SHELL) != 0) {
            exit(1);
        }
    }
}

```

当然，我们也可以让 `adbd` 保留 `root` 权限（在电脑端执行“`adb root`”命令，这会把系统属性 `service.adb.root` 设为 1），不过 `adb` 的源码中也对此功能做了一个限制——当系统属性 `ro.debuggable` 被设为 1 时，即使输入了“`adb root`”，`adbd` 也得不到 `root` 权限（同时还会输出一条我们很熟悉的出错消息：“`adbd cannot run as root in production builds`”）。系统属性 `persist.adb.trace_mask` 中存放了一个十六进制数，它表示的是一组指定日志输出详细程度的标志位（请试试把这个十六进制数设为 `0xff`，以获得最详尽的日志输出）。如果这个系统属性存在且有效，`adb` 会把日志记录到 `/data/adb/adb-%Y-%m-%d-%H-%M-%S` 文件中。

`adbd` 通常使用的是有 `init` 进程安装的 UNIX Domain socket——`/dev/socket/adb`，但是当设备通过 USB 线与电脑相连时（即，运行 `adbd` 的设备不是一个模拟器），它也会使用 `/dev/android_adb`（或者，从 L 版开始，改成 `functionfs` 伪文件系统中的文件 `/dev/usb-ffs/adb/ep##`）。后者是一个由 USB Gadget 驱动创建的设备节点。（在通过网络进行 `adb` 连接时）`adbd` 也会去监听系统属性 `service.adb.tcp.port` 中指定的 TCP 端口，如果这个系统属性不存在的话，该端口也可以由系统属性 `persist.adb.tcp.port` 指定。无论是上述哪种情况，`adb` 的架构都可以用图 5-1 来概括。

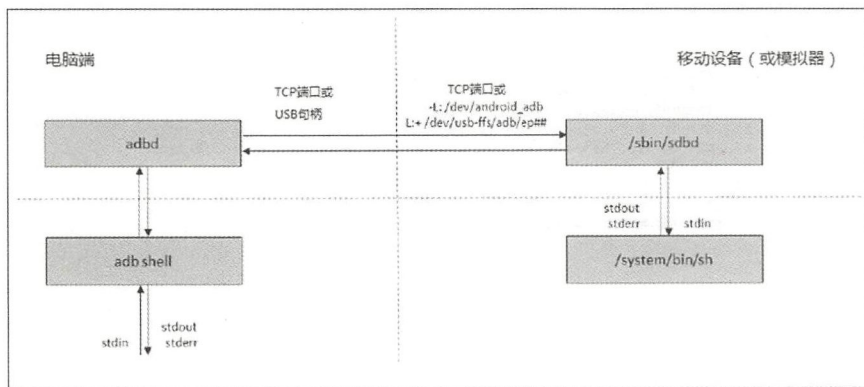


图 5-1 adb 的架构

## 跟踪 ADB 协议

ADB 所使用的协议已经详细地记录在其源码中的 protocol.txt 里了，所以这里也就没有必要再予以赘述了。ADB 通过环境变量 ADB\_TRACE，实现了一个非常简洁且有效的 trace 机制。该变量被传递给一个 adb 会话后，会让 adb 客户端程序输出详细的协议命令。在 adb 的源码中给出了该变量可以使用的一些参数（all、adb、socket、packet、rwx、usb、sync、sysdeps、transport、jdwp），但在实际使用过程中，在需要查看 ADB 消息时，transport 是最有用的参数之一——尽管它并没有给出消息发送/接收的方向（见输出结果 5-1）。

```
morpheus@Forge (~)$ ADB_TRACE=transport adb shell
30303063 000c                                # Message of length 12:
686f73743a766572736966e host:version
4f4b4159 OKAY                                # Reply
30303034 0004
30303166 001f
30303132 0012                                # Message of length 18:
686f73743a7472616e73706f72742d61 host:transport-a
4f4b4159 OKAY                                # Reply
30303036 0006                                # Message of length 6:
7368656c6c3a shell:
4f4b4159 OKAY                                # Reply
```

输出结果 5-1 使用 ADB\_TRACE=transport 列出 adb 协议命令

## ADB 的安全认证机制

由于 ADB 这个接口具有功能非常强大的调试和跟踪功能，因而万一它被非法使用，后果将不堪设想。尽管它是以 shell 这个 uid 的身份运行的，还远远没有 root 权限，但由于 shell 这个 uid 是好几个组（log、graphics 等）的成员，这也使得它具有很强的能力。使用 ADB 可以轻而易举地访问到用户的个人数据（包括开机图形锁的正确画法），也能把任意 App 或二进制可执行文件上传到设备上。出于这个原因，在 JellyBean 以后的 Android 系统中引入了公钥认证机制，把 ADB 的安全性又往上提升了那么一小步。相关的认证操作将会在系统属性 ro.adb.secure 的值被设为 1 时，通过 AUTH 消息完成（如代码清单 5-2 所示）。

AUTH 消息将作为对 OPEN 消息的响应，发送到电脑端，要求在其执行任何命令之前，先完成认证。AUTH 消息的参数总是一个 TOKEN，它是由移动设备中的随机数发生器（/dev/urandom）生成的一个大小为 20 个字节的随机数数组。移动设备端将等待电脑端用自己的私钥（该私钥应该已经生成，并存放在 \$HOME/.android/adbkey 这个文件中）对这个 TOKEN 进行签名，然后回复一个 AUTH SIGNATURE 消息，把用私钥加密（请把“加密”这个词读作：“签名”）过的随机数数组放在这个消息里返回给移动设备端。如果移动设备知道相应的公钥，那么验证就能继续，而且如果验证通过了，该会话就会切换到 online 状态上去。

因为所有这一切都依赖于公钥，这就产生了一个先有鸡还是先有蛋的问题：怎么让移动端事先就知道这个公钥，并将其用于验证呢？解决方案是允许电脑端响应一个 AUTH\_RSAPUBLICKEY 消息。因为此时这个公钥是不可信的，ADB 将把这个公钥通过/dev/socket/adb这个 UNIX Domain socket 传递给 system\_server（特别是由 com.android.server.usb.UsbDeviceManager 启动的 UsbDebuggingManager），然后，system\_server 将会弹出一个对话框（com.android.server.usb.UsbDebuggingActivity），要求用户确认该公钥的指纹（MD5 Hash）。如果用户选择信任该公钥，这个公钥就会被添加到 adb 的 key store（位于/data/misc/adb/adb\_key）中去。请注意：厂商可以轻而易举地重新编译 adb，删掉这一功能（相关代码位于 adb\_auth\_client.c 中的 db\_auth\_confirm\_key() 函数中），让 adb 只能使用事先写死的由厂商提供的公钥。

输入 dumsys usb 命令，你就能看到当前的 USB 调试状态（USB Debugging State）以及当前使用的 adb\_keys，如输出结果 5-2 所示。你有没有注意到它有点类似 SSH 的 known\_hosts 文件？这或许是某人灵光一现的结果吧！

```
shell@hammerhead:/ $ dumsys usb
...
USB Debugging State:
  Connected to adb: true
  Last key received: null
  User keys:
QAAAAAGih7j/oQP+S8AmUvBrpjxGY/5yppWThz4mpP6U9wt/fzGyip4sNt/2cp+40rRb8whQLALvPS2fAwLmlLj8TmJ/
... Public Key (as Base64)
+a+2cNPxxtmOh6GzOcnmwPaVsQcMLky1yCCS2o4hnjKYmjQBEAAQA= morpheus@Forge

  System keys:
IOException: java.io.FileNotFoundException: /adb_keys: open failed: ENOENT
(No such file or directory)
...
```

输出结果 5-2 用 dumsys usb 转储 USB 调试状态

## servicemanager

servicemanager 是 Android 中 IPC 机制的关键组件。这个二进制可执行文件尽管看上去很小，但却是非常重要的程序，没有它，系统内部进程间的通信就会受到很大的影响。其重要性从它在 /init.rc 中的定义就可窥见一斑，如代码清单 5-3 所示。



代码清单 5-3 servicemanager 在/init.rc 中的定义

```

service servicemanager /system/bin/servicemanager
class core
user system
group system
critical
onrestart restart healthd
onrestart restart zygote
onrestart restart media
onrestart restart surfaceflinger
onrestart restart inputflinger // Android L
onrestart restart drm

```

让 servicemanager 变得如此关键，有如此众多的服务有赖于它的原因是：它具有服务映射器（service mapper）的功能。事实上任何一种 IPC 机制都需要有这样一个映射器，好让客户端进程能够找到并连上服务端进程，UNIX 有它的 portmapper（供 sunrpc 使用），Windows 有它的 DCE endpointer mapper，而 servicemanager 则是 Android 中实现这一功能的组件。了解了这一点，/init.rc 中对 servicemanager 的定义就好理解多了——并不是这么多服务真的依赖 servicemanager，而是因为一旦 servicemanager 挂了的话，客户端进程就没法找到这些服务了。如果 servicemanager 重启了一下，它就会被刷回到一张白纸的状态——这就要求各个相关的服务重新去注册一下，以便能被客户端进程找到。因为对于服务进程来说，并没有什么机制能够让它们去随时检测 servicemanager 是不是挂了，所以唯一能让它们（在 servicemanager 重启之后）再去重新注册的方法就是：在 servicemanager 重启时，强制让这些服务也重启一下。

显然，servicemanager 及其所支持的各种框架服务理应得到更多的篇幅，而我把这个任务留给了后文：在下一章中，我们将把它和 system\_server（这个进程扮演着各个服务的宿主进程的角色）及各个独立的服务一起予以详细讨论。

## healthd

“健康度守护进程”（health daemon）的意思是：该服务会周期性地执行检测综合性的“设备健康值”的任务——尽管当前只有一个与电池电量有关的任务（这一点在将来发布的新版 Android 系统中或许会有所改变）。该守护进程也会把自己注册为 BatteryPropertiesRegistrar 服务（在 L 版中，则是 batterypropreg 或者 batteryproperties）。完成了这个注册之后，healthd 提供了几个框架服务（例如 BatteryStatsService），用它从 sysfs 那里得到的数据，更新电池电量信息。

和大多数守护进程一样，healthd 也会先去完成初始化配置，然后进入执行代码的循环。详细的过程如图 5-2 所示。

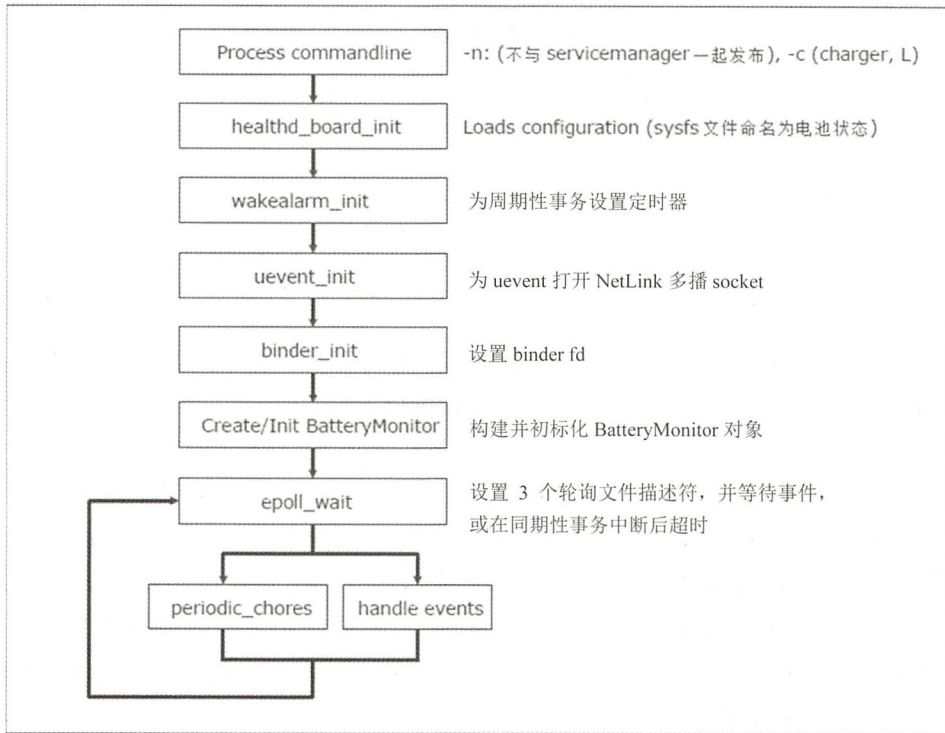


图 5-2 healthd 的执行流程

healthd 的主循环中使用 `epoll(2)` API 对三个文件描述符进行轮询操作（读），并在每次读取到数据时，执行已注册的相关操作，具体操作如表 5-1 所示。

表 5-1 healthd 使用的文件描述符及其作用

描述符	类 型	作 用
wakealarm_fd	TimerFD	该定时器（Timer）被设为每隔 <code>periodic_chores_interval</code> 秒就唤醒进程一次，在被这个定时器唤醒时，healthd 将会去执行 <code>periodic_chores</code> 的代码
event_fd	NETLink	读取内核通知事件。healthd 只会关心与其自身相关的 <code>power</code> 子系统（ <code>SUBSYSTEM=POWER</code> ）的事件。这些事件包括与电池电量及充电相关的通知。如果读取到了这些事件，healthd 进程将会去执行 <code>battery_update()</code> 函数
binder_fd	/dev/binder	（这种情况下，healthd 将以 <code>batteryprog</code> 服务的身份出现）让各个框架客户端中的 Listener 更新相关信息

第一个要读取的文件描述符是 `wakealarm_fd`，healthd 用它执行周期性的日常工作。这里一共使用两种内部类型：`fast`（1 分钟，用在使用电源充电时）和 `slow`（10 分钟，用在使用电池供

电时)<sup>1</sup>。当前定义的唯一一个日常工作是 `battery_update()`，该函数会让 `healthd` 以 `BatteryPropertiesRegistrar` 服务的身份，更新电池电量的当前状态。在从 `event_fd` 这个 NETLink 接收到来自 POWER 子系统的事件时，这个函数也会被调用：`healthd` 并不会去解析这些事件中的具体信息，只是去刷新一下电池电量的当前状态。之所以需要后一种模式，是因为 `healthd` 需要能够对插上/拔下充电器之类的事件，或是其他电源管理警告做出响应。最后那个 `binder_fd` 是供框架中的 listener（首先是 `BatteryStatsService`）使用的接口，我们将在下一章中予以讨论。

## 实验：观察 `healthd` 的后台行为

使用功能强大的 `strace(1)` 实用程序，你可以观察到 `healthd` 在后台的行为：通过附加 `healthd` 进程的 ID（需有 root 权限）并调用 `ptrace(2)` API，`strace(1)` 可以在 `healthd` 调用各个系统调用的同时得到相关通知（见输出结果 5-3）。因为任何一个进程只要想做一点真正有意义的事，都必须使用系统调用，这就向我们提供了一种能够详细地跟踪 `healthd` 行为的方法，并揭示出 `healthd` 是通过 `sysfs` 伪文件系统中的哪些伪文件来获取电池电量使用情况的。

```
root@htc_m8w1:~# ls -l /proc/$healthd_pid/fd | cut -c'1-10,55-'
lrwx----- 0 -> /dev/null
lrwx----- 1 -> /dev/null
lrwx----- 2 -> /dev/null
l-wx----- 3 -> /dev/_kmsg_ (deleted)           # Output: Log to kernel
lrwx----- 4 -> socket:[6951]                   # event_fd (NetLink socket)
lrwx----- 5 -> /dev/binder                     # binder fd
lrwx----- 6 -> anon_inode:[eventpoll]         # epollfd
l-wx----- 7 -> /dev/cpuctl/apps/tasks           # fg cgroup fd (libcutils)
l-wx----- 8 -> /dev/cpuctl/apps/bg_non_interactive/tasks # bg cgroup fd (libcutils)
lr-x----- 9 -> /dev/_properties                 # r/o property fd

root@htc_m8w1:~# strace -p $healthd_pid
Process $healthd_pid attached - interrupt to quit
# healthd patiently polling (0xffffffff = indefinitely) until an fd signals an event
epoll_wait(0x6, 0xb5bb5898, 0x2, 0xffffffff) = 1
# NetLink msg received on fd 4 (event fd) - indicating core state change (going offline)
recvmsg(4, {msg_name(12)={sa_family=AF_NETLINK, pid=0, groups=00000001},
msg_iov(1)=[{"offline#/devices/system/cpu/cpul"... , 1024}], msg_controllen=24, ...
# healthd's not interested, so it goes back to polling
epoll_wait(0x6, 0xb5bb5898, 0x2, 0xffffffff) = 1
# message indicating change in battery status:
recvmsg(4, {msg_name(12)={sa_family=AF_NETLINK, pid=0, groups=00000001},
msg_iov(1)=[{"change#/devices/platform/htc_bat"... , 1024}], msg_controllen=24,
{cmsg_len=24, cmsg_level=SOCK_CMSG, cmsg_type=SCM_CREDENTIALS{pid=0, uid=0, gid=0},
msg_flags=0}, 0) = 488
#
# healthd goes into a flurry of statistics collection, opening and closing files:
#
open("/sys/class/power_supply/battery/present", O_RDONLY) = 10    # Is battery present?
read(10, "\n", 16)        = 2                                     # Yes (1)
close(10)                 = 0
open("/sys/class/power_supply/battery/capacity", O_RDONLY) = 10    # What is its capacity?
read(10, "96\n", 128)     = 3                                     # 96%
close(10)                 = 0
open("/sys/class/power_supply/battery/batt_vol", O_RDONLY) = 10    # Voltage?
read(10, "4303\n", 128)   = 5
close(10)                 = 0
...
open("/sys/class/power_supply/wireless/online", O_RDONLY) = 10    # Alas, no wireless charging
read(10, "0\n", 128)      = 2                                     # for the MS
close(10)                 = 0
write(3, "<C>healthd: battery l=96 w=4 t=2*... , 51) = 51         # Report to kernel log
ioctl(5, BINDER_WRITE_READ, 0xb5bb5070) = 0                     # Report to client listeners
epoll_wait(0x6, 0xb5bb5898, 0x2, 0xffffffff) = ..                # Back to polling
```

输出结果 5-3 使用 `strace(1)` 分析 `healthd` 的行为

1 有意思的是，在许多 Android 发布版本中，调用 `timerfd_create` 会返回一个 `EINVAL`（参数无效），而不会创建 `wakealarm_fd`，因此也导致读取 `event_fd` 成了唯一的事件源。——原注



注意：sysfs 中的伪文件（/sys/class/power\_supply/\*）的名称都是标准的——事实上，它们都是指向特定的平台设备节点的符号文件，而这些设备节点的名称，在不同的设备上可能是各不相同的。

我们还可以进一步改进上述操作步骤：你可以（按下&键）把 strace 切换到后台去，然后拔掉 USB 连接线，然后再把它插上。这时，你会看见报告电池状态改变的 NetLink 通知消息，后面还会跟着一个/sys/class/power\_supply/usb/online 文件内容改变的通知（从 1 变成 0 表示断开连接，反之则表示连上了 USB 线）。

从 L 版开始，healthd 也支持 dumphsys 了，你可以从 Android L 系统中把 dumphsys 这个二进制可执行文件 adb pull 出来（从一台 Nexus 5 手机或是模拟器中），然后再把它 adb push 到你自己的手机里并执行之，这个程序的执行结果如输出结果 5-4 所示。

```
# Before: Only KK healthd - note old service name (batterypropreg)
#
root@htc_m8wl:/ # service_list | grep batteryprop
91 batterypropreg: [android.os.IBatteryPropertiesRegistrar]
root@htc_m8wl:/ # /data/local/tmp/healthd.L & # Run healthd from L
[1] 7287
# After: new service name (batteryproperties) added. Name is different, so no conflict
#
root@htc_m8wl:/ # service_list | grep batteryprop
0 batteryproperties: [android.os.IBatteryPropertiesRegistrar] # L: diff. name, same iface
92 batterypropreg: [android.os.IBatteryPropertiesRegistrar]
root@htc_m8wl:/ # dumphsys batteryproperties # Calling dumphsys
ac: 0 usb: 1 wireless: 0
status: 5 health: 2 present: 1
level: 100 voltage: 4 temp: 273
```

输出结果 5-4 在一台运行 KitKat 系统的设备上执行 L 版中的 dumphsys 的结果

如果你同时还使用着 strace，你还能够看见 dumphsys 在后台的行为。如输出结果 5-5 所示（因为这里的文件描述符不一定和你机器上的一致，所以我用符号代替了它们）。

```
epoll_pwait(epoll_fd, {{EPOLLIN, {u32=37597, u64=12884939485}}}, 2, -1, NULL) = 1
ioctl(binder_fd, BINDER_WRITE_READ, 0xbeab1748) = 0 # Incoming binder req
write(...tasks, healthd_pid, 4) = 4 # Make healthd foreground
..
write(new_fd, "ac: 0 usb: 1 wireless: 0\n", 25) = 25 # Write output
write(new_fd, "status: 5 health: 2 present: 1\n", 31) = 31 # to binder supplied
write(new_fd, "level: 100 voltage: 4 temp: 273\n", 32) = 32 # file descriptor.
fsync(new_fd) = -1 EINVAL (Invalid argument)
ioctl(binder_fd, BINDER_WRITE_READ, 0xbeab1600) = 0
close(new_fd) = 0
write(...tasks, healthd_pid, 4) = 4 # Make healthd background
..
ioctl(binder_fd, BINDER_WRITE_READ, 0xbeab1758) = 0
```

输出结果 5-5 在执行输出结果 5-4 时，同时运行 strace 的结果

尽管这个实验中并没有详细给出 L 版 Android 系统中 healthd 的内部工作机制，但也展示了

Android 的一个重要的部分：通过 binder 进行进程间通信（IPC）。在上文中，你可以看见一个文件描述符是如何通过主调进程（`dumpsys`）传递给 `healthd` 的。Binder 的内部工作机制非常复杂，我将把详细讨论 Binder 的任务留给本系列的第 2 本<sup>1</sup>。

## 被用作 charger 守护进程的 `healthd`

在 L 版之前的 Android 系统中，有一个特殊的守护进程 `charger`，如果系统检测出自己正在充电模式下启动，那么 `init` 进程就会启动这个守护进程（通过 `class_start charger` 指令，仅包含一个单独的服务）。在 L 版中，`charger` 已经被合并到 `healthd` 里去了——这么做有一定的道理，因为反正 `healthd` 的主要任务就是监视电池电量的相关状态，在以 `charger` 模式运行时，`healthd` 会以一种类似之前讨论过的 `init` 的“分身”（`watchdog` 和 `ueventd`）的形式运行。换言之，`/bin/charger` 现在只不过是 `/sbin/healthd` 的一个符号链接，只不过是用一个 `-c` 参数启动的罢了。`charger` 守护进程负责在设备充电时，把电池电量信息以图形的形式显示给用户看。这是使用 `MinUI` 库（我们将在第 2 本中详细讨论它）来实现的。

尽管只是一个猜测，但是从 L 版开始，`healthd` 可能会被加入更多的功能，并在 Android 中扮演一个更为重要的角色。一个暗示了它的重要性的迹象是：除了它还是一个关键服务（用 `critical` 关键字修饰）之外，它也是为数不多的几个被放在 `root` 文件系统上的守护进程之一（`healthd` 位于 `/sbin` 目录中，而不像其他大多数守护进程那样位于 `/system/bin` 目录中）。

## Imkd (Android L)

Android L 版中使用了一个特殊的 `core` 类的服务类守护进程 `lmkd`。它在 `/init.rc` 中的定义如代码清单 5-4 所示。

代码清单 5-4 `lmkd` 在 `/init.rc` 中的定义

```
service lmkd /system/bin/lmkd
    class core
    critical
    socket lmkd seqpacket 0660 system system
```

`lmkd` 提供了内核中的 LMK（Low Memory Killer）机制<sup>2</sup>的一个接口，这是个 Android 特有的东西（也就是说，它是个只存在于 Android 内核中、Linux 中没有的特性）。`lmkd` 允许 Android 高级用户能够（部分）控制 Linux 的 Out-Of-Memory(OOM)机制[即，在系统物理内存不足时，

1 在本书的下一章中也会对 Binder 进行部分讨论。——译者注

2 本书第 7 章中有 LMK 的相关讨论。——译者注

通过杀掉 (kill) 部分进程的方式, 缓解内存压力的机制]。lmkd 可以使用 `/proc/pid/oom_score_adj` 文件, 通过调整相关进程的 OOM 分值的方式, 使它们在系统面临内存压力时, 更容易或更不容易被杀掉。Linux 的 OOM 机制我们会在后续的书予以讨论。lmkd 有两种可能的操作模式——具体进入哪种模式, 是根据是否检测到了 LMK 这一 Android 特性而决定的。如果检测到了 LMK, lmkd 将只去调整目标进程在 `/proc` 伪文件系统中的 OOM 分值, 而把系统面临内存不足的压力时具体该杀掉哪个进程的任务留给 LMK 模块。如果内核中没有 LMK, lmkd 也会担负起响应内存压力事件的任务, 并具体执行杀进程的任务 (也就是向被杀的进程发送 SIGKILL 信号)。这时, lmkd 会维护一张进程的 Hash 表, 以便快速查阅所有进程的 OOM 分值。

和本章中讨论的其他守护进程一样, lmkd 也会使用 `epoll_wait` 来同时等待多个 socket 的输入。该进程主要监听的 socket 是 `/dev/socket/lmkd`, 它是由 `init` 进程为 lmkd 创建的, 用来监听连接的。这个 socket 唯一期望连接的客户端是 `ActivityManagerService` (将在下一章详细讨论), 该服务通过这个 socket 来通知 lmkd 哪个进程需要调整它的 OOM 分值 (通过 `ProcessList` 类)。当内核中 LMK 不可用时 (即, 找不到 `/sys/module/lowmemorykiller` 文件时), lmkd 还会额外去监听 `/dev/memcg` 这个 cgroup 中的文件, 以获取内存压力事件, 整个流程如图 5-3 所示。

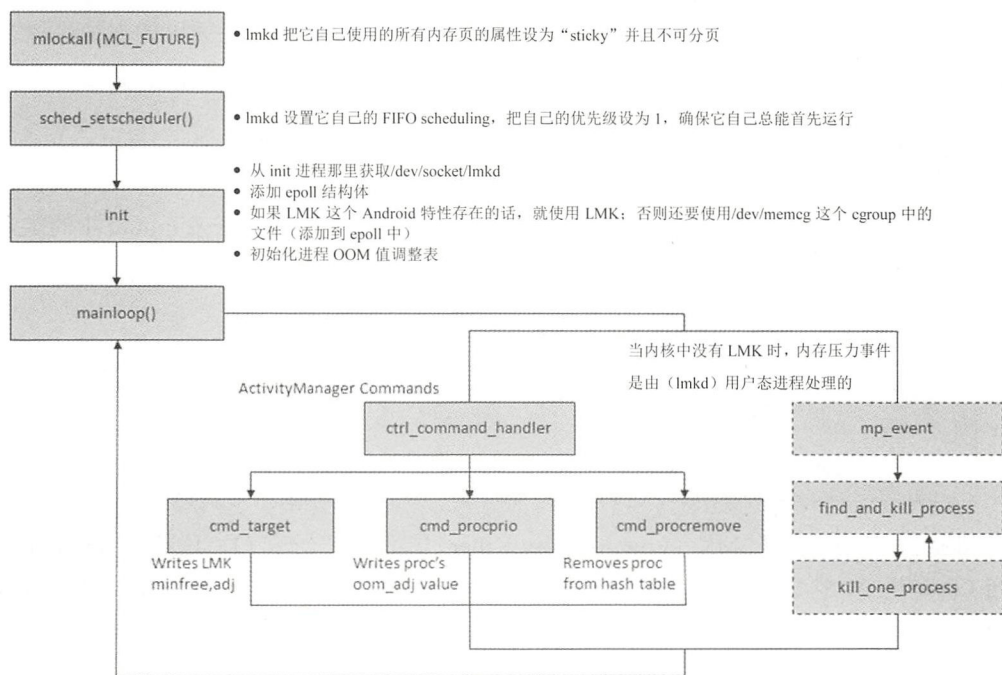


图 5-3 lmkd 的执行流程



在负责处理内存压力事件时 [也就是在内核中没有使用 LMK (low memory killer) 的情况下], `lmkd` 会去解析 `/proc/zoneinfo` 文件, 获取下列值:

- **nr\_free\_pages**——空闲内存的总数 (单位: 4KB)。
- **nr\_file\_pages**——文件映射占用的内存总数 (单位: 4KB)。
- **nr\_shmem**——共享的内存的总数, 也就是由多个进程公用的内存页, 因而也应该从因文件映射而占用的内存总数中逐次<sup>1</sup>减掉。
- **nr\_totalreserve\_pages**——系统保留的内存的总数, 尽管这些页是空闲的, 但它们实际上却是不能使用的, 所以也应该从空闲内存总数中减掉。

然后 `lmkd` 会不断地杀掉被选中的进程, 直到空闲内存总数达到要求或目标文件能被映射到内存中去为止。

### 实验: 观察 `lmkd` 的后台行为

在本节写作的时候, Android L 版的源码还没有被发布出来 (只有一个非常有限的预览版)。不过已经可以从 Nexus 5 或模拟器那里获得 Android L 版的系统镜像了。因此也就可以很方便地使用逆向工程的方法, 获取足够写作本节所需的细节信息。我同时使用了静态分析 (反汇编) 和动态分析 (运行时调试) 的方法。上一节中 (使用 `strace`) 分析 `healthd` 的方法, 在这里又一次证明了它的有效性 (见输出结果 5-6)。注意: `lmkd` 是不能被复制到 KitKat (或之前的版本) 中去使用的, 因为它必须依靠 `init` 为它创建的 `seqpacket socket` 才能与其他框架进行通信。

```
root@LEmulator:/# ls -l /proc/$lmkd_pid/fd | cut -c1-10,55-
lrwx----- 0 -> /dev/null
lrwx----- 1 -> /dev/null
lrwx----- 10 -> socket:[7360]          # /dev/socket/lmkd (listening)
lrwx----- 2 -> /dev/null
lrwx----- 3 -> anon_inode:[eventpoll]
lrwx----- 4 -> socket:[7364]          # /dev/socket/logdw (to logd)
lrwx----- 5 -> socket:[7653]          # /dev/socket/lmkd (to ActivityManager)
lr-x----- 8 -> /dev/_properties_
root@LEmulator:/# strace -p $lmkd_pid
epoll_pwait(3, {{EPOLLIN, {u32=3069216345, u64=37428954713}}}, 2, -1, NULL, 8) = 1
read(5, "\0\0\0\1\0\0\4\5\0\0\0\0\0", 52) = 12
openat(AT_FDCWD, "/proc/1029/oom_score_adj", O_WRONLY) = 6
write(6, "547", 3) = 3
close(6) = 0
```

输出结果 5-6 使用 `strace` 分析 `lmkd` 的后台行为

1 这个“逐次”的意思是: 如果一个页被  $n$  个进程共享, 那么就应该被减掉  $n-1$  次。——译者注

从上面这个输出结果我们可以看到：lmkd 与其他的守护进程一样，是用 `epoll_wait`（FD 为 3）阻塞掉自己，等待事件发生的。用来输入的 fd（即 5），就是 `/dev/socket/lmkd`，这个 socket 的另一端连接的是 Android 的 `ActivityManagerService`，所传递的消息长度可变（最多不超过 52 个字节），并以某个消息类型开头。我们可以观察到 3 种消息类型，如表 5-2 所示。

表 5-2 lmkd 协议使用的消息

常 量	类 型	参 数
LMK_TARGET	0x00000000	将 lmkd 要写入 <code>/sys/module/lowmemorykiller/parameters/minfree</code> 的整型数组作为参数
LMK_PRIO	0x00000001	要调整 OOM 的进程的 PID（比如上面这个输出结果中的“\0\0\4\5”表示的就是 PID 1029 <sup>1</sup> ）及要设置的 <code>oom_score_adj</code> 值
LMK_PROCREMOVE	0x00000002	需要移出监视名单的进程的 PID

## logd（Android L）

Android L 版中新增了一个需求呼声很高的日志机制，它是由 `logd` 守护进程来实现的。相对于旧版 Android 使用 `/dev/log` 中的各个文件（这些文件是由内核中的各个对应的 `ring buffers` 实现的）记录日志的方式，这个守护进程提供了集中式的用户态日志记录服务。这不光解决了 `ring buffer` 的主要缺点（它们太小而且需要常驻内存），而且还让 `logd` 能和 SELinux 的审计操作结合在一起——通过将自身注册为 `auditd`，就能从内核中（通过 `netlink`）接收到 SELinux 消息，并把它们记录到系统日志里去。

`logd` 提供的另一个重要的新特性是“日志修剪”（`log pruning`），这一特性允许自动清理或保留指定 UID 的日志记录。这一特性的设计目标是解决因为某些进程会记录过于详细的日志，而把整个日志全给“淹没”掉了的情况发生——在这种情况下，要把有意义的日志从大量垃圾信息里分离出来，是个很难完成的任务。使用 `logcat` 新增的 `-P` 参数，可以给 `logd` 设置“白名单”（必须记录其生成的消息的 UID 或 PID 列表）或是“黑名单”（需要忽略其生成的消息的 UID 或 PID 列表）。

`logd` 服务在 `/init.rc` 中的定义如代码清单 5-5 所示。

1 这里显然有点问题，十进制的 1029 对应的十六进制数是 0x405。——译者注



代码清单 5-5 logd 服务在/init.rc 中的定义

```
service logd /system/bin/logd
    class core
    socket logd stream 0666 logd logd      # Used by CommandListener thread
    socket logdr seqpacket 0666 logd logd  # Used by LogReader thread
    socket logdw dgram 0222 logd logd      # Used by LogListener thread
    seclabel u:r:logd:s0
```

请注意，和这个服务配套的 socket 不是一个，而是四个：

- **/dev/socket/logd**——该 socket 用作控制接口。
- **/dev/socket/logdw**——这是一个只读的 socket（权限设置为 0222 = -w--w--w-）。
- **/dev/socket/logdr**——这是一个读写 socket，设计初衷是用来读。它是个 seqpacket（sequential packet）socket，而不是 UNIX domain socket。
- 一个 **unnamed** 的 **NetLink socket**——在 logd 提供 auditd 功能时，供 SELinux 使用。

logd 会为它的各个 socket 创建监听线程（listener thread），同时也会在各个客户端（发起请求时）创建专门的线程（处理相应的请求）。这些线程都是独立命名的（使用 prctl(2)），所以你可以在 logd 正在运行的时候，在 logd 的 /proc/\$pid/task 目录中亲眼看到这些线程。

和传统的各个 log 文件一样，logd 也能认出 main、radio、events 和 system 这几个 log buffer，此外，在 L 版中还新增了一个 log，即 crash。这些 log buffer 可以用它们的“log id”（lid）来表示，依次编号为 0 到 5。

## logd 使用的系统属性

logd 所能识别的数个系统属性，均位于命名空间 logd 之下，这些属性的值决定了 logd 的行为。在 logd 目录中的 README.property 文件中对这些属性做了详细的文档说明，不过为了方便（读者查阅）起见，我还是把它们列在了这里（见代码清单 5-6）。

代码清单 5-6 logd 使用的系统属性

name	type	default	description
logd.auditd	bool	true	Enable selinux audit daemon
logd.auditd.dmesg	bool	true	selinux audit messages duplicated and sent on to dmesg log
logd.statistics.dgram_glen	bool	false	Record dgram_glen statistics. This represents a performance impact and is used to determine the platform's minimum domain socket network FIFO size (see source for details) based on typical load (logcat -S to view)
persist.logd.size	number	256K	default size of the buffer for all log ids at initial startup, at runtime use: logcat -b all -G

# persist.logd.size.logname can be used to set buffer sizes for individual logs





控制 logd

客户端程序可以连到/dev/socket/logd 上，用一组协议命令来控制 logd。通常这样做的客户端程序是 logcat 命令。这个程序的代码也已经被改过了，它不再使用 ioctl(2)直接操作/dev/log/目录下各个文件的方式，而是改用通过 socket 与 logd 通信的方式实现相关功能了。（通过 socket 发送的）相关命令如表 5-3 所示。

表 5-3 logd 的协议命令

命 令	logcat 参数	作 用
clear lid	-c	如果调用者有权进行该操作，这条命令将清空指定的 log buffer
getLogSize lid	-g	获取由 lid 参数指定的 log buffer 的最大尺寸
getLogSizeUsed lid		获取由 lid 参数指定的 log buffer 的当前已用大小
setLogSize lid	-G	设定由 lid 参数指定的 log buffer 的最大尺寸
getStatistics lid	-S	如果调用者有权进行该操作，提取出日志中的相关记录，比如可以列出指定 PID 的日志等
getPruneList	-p	获取（所有 log 的）prune 列表
setPruneList	-P	设置（所有 log 的）prune 列表
shutdown		强制守护进程退出，不过令人惊奇的是，这一操作竟然不需要任何权限认证

灰色的命令需要主调进程拥有进行 log 操作的权限（有 root 权限），拥有 root、system 或 log 的主组（primary group）的 GID，或有 log 的次级组的 GID。在验证最后那种情况时，logd 中的代码做了一个很粗暴的操作，即直接解析主调进程的/proc/pid/status 文件中“Group:”一行中的内容。

向 logd 写入数据（记录日志）

Android 的 log 机制是由 liblog 支持的，因此应用可以名正言顺地无视 liblog 之下的 log 机制的具体实现。截至 L 版，Bionic 和 liblog 都可以编译成可以用于 logd 的形式（使用宏定义 TARGET\_USES\_LOGD）——这会让所有与 log 相关的 API 都改而使用 logd，而不是像以前那样使用/dev/log 中的各个设备文件。事实上，/dev/log 已经成为了历史（而且显然已经在 M 版中被去掉了）。实施这一改变需要做出的改动很小，因为系统中所有与 log 相关的 API，最终都要去调用 liblog 中的\_\_android\_log\_buf\_write（或 Bionic 中的\_\_libc\_write\_log），然后这个函数会去打开 logdw 这个 socket（它代替了/dev/log），并把 log 消息写入这个 socket。图 5-4 显示了把应用中的 log 消息写入 logd 的所有路径。它与事件日志（Android.util.EventLog）消息的写入流程十分相似。

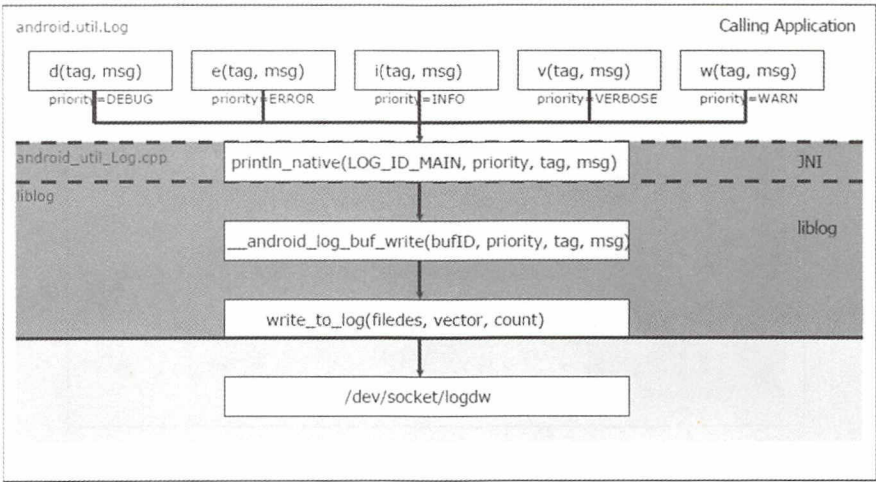


图 5-4 Android 的 log 架构

从 logd 中读取日志（logcat）

在 L 版中，我们熟悉的 logcat 仍然支持之前版本中使用的所有命令行参数。不过，在底层实现中，它已经被重写过了一—使用新版 liblog 中的一个 API，改用 logd 了。logcat 之类的客户端可以连到 logd 的数据读取 socket（/dev/socket/logdr），并通过向 logd 中的 LogReader 类实例写入一些参数的方式，要求它提供日志数据。相关参数如表 5-4 所示。

表 5-4 logd 通过数据读取 socket 接受的参数类型

参 数	含 义
libs = value	要求提供指定 log id 的日志
start = value	获取从指定时间开始的日志（默认值是 EPOCH，也就是日志开始记录的时间）
tail = value	获取指定行数的日志（类似于 tail(1)命令）
pid = value	获取进程 id 为指定 pid 的进程产生的日志
dumpAndClose	告诉日志数据读取线程，在获取到（dump）相关日志数据后退出

日志记录会被顺序打包到 `logger_entry_v3` 结构体中，然后再通过 socket 传递给要求读取日志数据的客户端。`logger_entry_v3` 结构体的格式如图 5-5 所示。

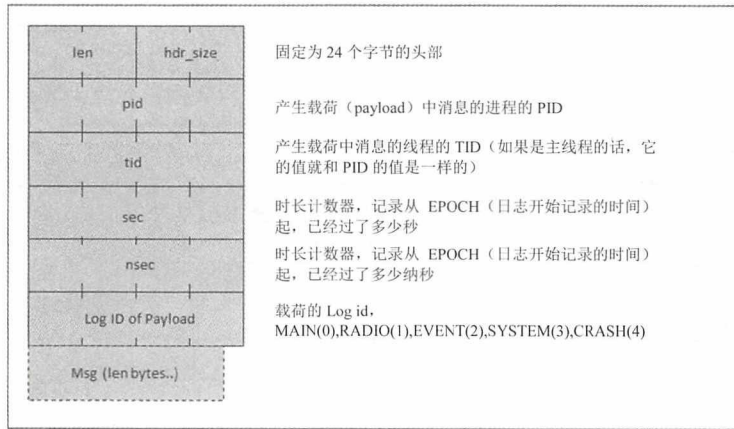


图 5-5 logd 消息的格式

了解了上述这些知识后, 我们现在可以通过 logcat 命令, 实时地观察 logd 的行为, 我们来看下面这个实验。

### 实验：观察 logcat

strace 让你可以观察到 logcat 没有在界面上显示出来的行为, 包括它是怎样连上 logd 的, 它在需要获取日志数据时发送的命令, 以及已经顺序打包完毕的日志消息 (见输出结果 5-7)。

```
# Tracing logcat during an adb logcat operation shows messages are received
# from file descriptor 3, and sent to file descriptor 1 (stdout)

root@generic:/# strace logcat
...
connect(3, {sa_family=AF_LOCAL, sun_path="/dev/socket/logdr"}, 20) = 0
write(3, "stream lids=0,3,4", 17) = 17 # Dump main, system, crash
...
# \16 = 14 bytes (payload). \30 = 24 bytes (header). T\1 = 340 (PID) ... \3\0\0\0 = System
recvfrom(3, "<\16\30\0T\1\0\0m\1\0\0\275bbT$+\2374\3\0\0\0\6Act".., 5120, 0, NULL, 0) = 3668
write(1, "E/ActivityManager( 340): ANR in...", 5472) = 5472
# In case you missed the connect(2) call above (e.g. if attaching to logcat), you can still
# look through its /proc/..fd entry, to see file descriptor 3 is a socket - which you can
# also deduce from the use of rcvfrom(2):
root@generic:/# cd /proc/$LOGCAT_PID/fd
root@generic:/proc/337/fd # ls -l | grep "3"
lrwx----- root root 2014-11-11 14:24 3 -> socket:[2442]
# Looking through /proc/net/unix, which shows domain sockets, we can find the socket
and its remote endpoint (next inode number) - which happens to be logdr
root@generic:/proc/337/fd # grep 2442 /proc/net/unix
00000000: 00000003 00000000 00000000 0005 03 2442
root@generic:/proc/337/fd # grep 2443 /proc/net/unix
00000000: 00000003 00000000 00000000 0005 03 2443 /dev/socket/logdr
```

输出结果 5-7 strace 监视下的 logcat 行为 (带注释)

切换到 logd 进程那里, 找到并跟踪 (trace) logd.reader.per 线程实例, 将会从 logd 的视角向你展示日志的实现过程, 我就把这个实验留给读者你吧!





## vold

Android 中的 vold 是卷管理（volume-management）守护进程。这个概念 [使用一个用户态中的守护进程，在内核检测到文件系统（卷）时，自动 mount 它们] 最初是源自 Solaris 操作系统的（尽管现在在 Solaris 操作系统上已经没有 vold 了）。从 HoneyComb 版起，vold 也开始支持文件系统加密，特别是对/data 分区的加密。vold 在/init.rc 中的定义，如代码清单 5-7 所示。

代码清单 5-7 vold 在/init.rc 中的定义（系统版本 KitKat）

```
service vold /system/bin/vold
class core
socket vold stream 0660 root mount
ioprio be 2
```

vold 是唯一一个被表明 ioprio 等级（用来指定服务的 I/O 优先级）的守护进程。

vold 和 init 共享了一段通用的代码——fs\_mgr，这段代码被同时以静态编译的形式，编译进了这两个可执行文件。fs\_mgr 中提供了 mount 和检查文件系统的功能——其实现方式既有封装系统调用（比如 mount(2)），也有以预先写死的代码，使用其他二进制可执行文件（/system/bin/e2fsck）。

## 配置文件

要成为一个能真正完成 mount 卷任务的守护进程，vold 需要一个配置文件——在其中列出所有已知的文件系统以及它们的 mount 点。这个文件也被称为 **file system table**（文件系统表），或简称为“**fstab**”。在 Android 4.3 版以前，这个文件是/system/etc/vold.fstab，并根据文件系统在/sys 中的块设备（block device）所在路径表示各个分区。不过，从 4.3 版起，这个文件已经被挪到 rootfs 中去了，指定要 mount 哪些文件系统的配置文件，也像设备中的其他一些.rc 文件一样，被重命名为/fstab.\${ro.hardware}的格式。数据的存储格式也与经典 UNIX 系统中的 fstab 文件中使用的一致，如代码清单 5-8 所示。

代码清单 5-8 Android 4.3 版后/fstab.\${ro.hardware}使用的语法

```
# Android fstab file.
# The filesystem that contains the filesystem checker binary (typically /system) cannot
# specify MF_CHECK, and must come before any filesystems that do specify MF_CHECK

#<src>          <mnt_point>    <type>    <mnt_flags and options>
/devices/mmc.2/mmc_host    auto        vfat      defaults        voldmanaged=ext_sd:auto,noemulatedsd
/devices/platform/xhci-hcd  auto        vfat      defaults        voldmanaged=usb:auto
```

你可能还记得，（在上一章）讨论/init 是如何 mount 文件系统时，我们也遇到过 /fstab.\${ro.hardware} 文件，vold 解析该文件的方式与 init 使用的方式是一样的（它们用的都是



fs\_mgr 中的同一段代码)，不过 init 会忽略掉那些带有 voldmanaged 的行，而只有 vold 才会去关注这些行。mnt\_flags 字段中的内容（尽管在文档中被错误地注明会被忽略）会被一字不差地传递给系统调用 mount(2)，fs\_mgr 所能解析的可选参数如表 5-5 所示。

表 5-5 fs\_mgr 使用的可选参数

可选参数	作 用
wait	为了保证文件系统已经 mount 完毕，在继续之前，等待 20 秒
check	在 mount 之前，检查一下文件系统
nonremovable	表示这个卷是个不可移动的卷（也就是说不是一张 SD 卡）
recoveryonly	只有在进入 recovery 模式时才 mount 这个文件系统
noemulatedsd	告诉系统，这不是一个模拟 SD 卡，如果它是个 vfat 格式的文件系统，就要使用 ASEC
verify	从 KitKat 版开始才有这个参数：启用 Linux 内核中的 dm_verity 功能，以密码学的方式校验文件系统的完整性（详见第 8 章）
zramsize=	压缩 RAM（ZRAM）的大小
swapprio=	该分区被用作系统 SWAP 分区的优先级
lenth=	表示分区的大小
voldmanaged=	该分区是由 vold 管理的。等号后面跟的应该是 tag:number[其中的 number 是分区号(partition number) 或 “auto”] 这种形式
encryptable=	指定加密文件系统的解密密钥的存储位置
forceencrypt	L 版中新增：在 first boot 时就使用加密

内部架构

vold 内部可以分为三个组件。

- **VolumeManager**: 负责维护卷的状态，并处理各种卷操作，这个（单独的）类提供了所有面向框架（framework-facing）的功能。
- **NetLinkManager**: 负责监听 block 子系统使用 NetlinkHandler 传递给 VolumeManager 的内核 NetLink 事件。
- **CommandListener**: 负责监听/dev/socket/vold 这个 socket，它是用来接收框架(framework) 发出的命令，并传递这些命令的执行结果，或其他从 VolumeManager 那里接收到的事件。

图 5-6 中给出了 vold 的组成结构。

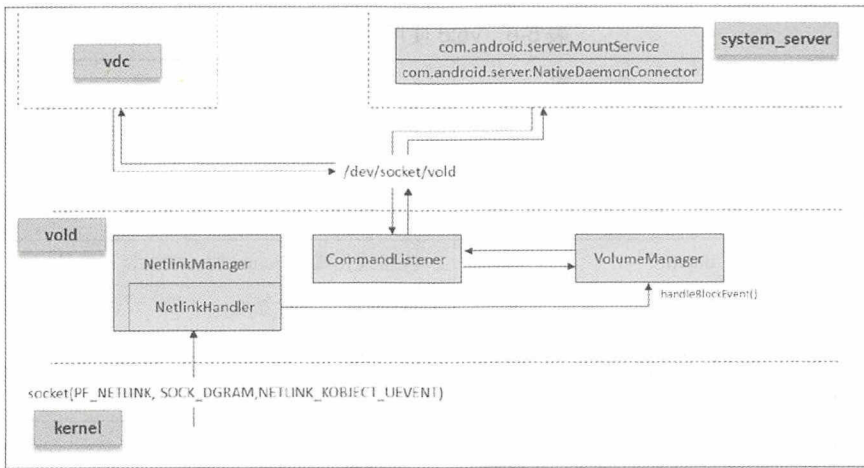


图 5-6 vold 的内部架构

vold 的主要客户端是 `com.android.server.MountServices`，不过应用是不能直接调用这个服务的，必须代之以 `android.os.storage.StorageManager`。`MountServices` 使用一个名为 `NativeDaemonConnector` 的类，这个类中的代码会连上 `/dev/socket/vold` 这个 socket，通过它向 vold 中的 `CommandListener` 发送命令。大多数 Android 设备上都有一个 vdc 实用程序，你可以用它亲手向 vold 发送这些命令（需要 root 权限），或者像输出结果 5-8 所示的那样，监听与文件系统 mount 相关的事件（使用 `vdc monitor`）。

```
root@htc m8wl:/ # vdc monitor
[Connected to Vold]
# SD-Card inserted
605 Volume ext_sd /storage/ext_sd state changed from 0 (No-Media) to 2 (Pending)
605 Volume ext_sd /storage/ext_sd state changed from 2 (Pending) to 1 (Idle-Unmounted)
630 Volume ext_sd /storage/ext_sd disk inserted (179:128)
630 Volume ext_sd /storage/ext_sd disk inserted (179:128)
605 Volume ext_sd /storage/ext_sd state changed from 1 (Idle-Unmounted) to 3 (Checking)
613 ext_sd /storage/ext_sd "8A07-A343"
614 ext_sd /storage/ext_sd
605 Volume ext_sd /storage/ext_sd state changed from 3 (Checking) to 4 (Mounted)
# SD-Card removed
632 Volume ext_sd /storage/ext_sd bad removal (179:129)
605 Volume ext_sd /storage/ext_sd state changed from 4 (Mounted) to 5 (Unmounting)
613 ext_sd /storage/ext_sd
614 ext_sd /storage/ext_sd
605 Volume ext_sd /storage/ext_sd state changed from 5 (Unmounting) to 1 (Idle-Unmounted)
631 Volume ext_sd /storage/ext_sd disk removed (179:128)
605 Volume ext_sd /storage/ext_sd state changed from 1 (Idle-Unmounted) to 0 (No-Media)
```

输出结果 5-8 使用 vdc monitor 监听到的 mount SD 卡的相关事件

实用程序 `vdc` 不过是一个 UNIX domain socket 的瘦客户端罢了，它的源码可以在 `system/vold/vdc.c` 中找到。这些会被一字不差地传递给 vold 的命令，如表 5-6 所示。





表 5-6 vold 使用的命令

命 令	子命令	参 数	作 用
dump			dump 出所有 loop mount 的 device mapper 以及已经 mount 上来的文件系统
volume	list		列出所有已经 mount 了的卷
	debug	on off	在格式化或 unmount 时，触发调试消息
	mount	<i>path</i>	mount 一个文件系统
	unmount	<i>path</i> [force[_and_revert]]	unmount 一个文件系统，可以强行 unmount
	[un]share	ums	共享/不共享大容量 USB 存储设备
	shared	ums	返回大容量 USB 存储设备的共享状态(enabled/disabled)
	mkdirs	<i>path</i>	创建一个 mount 点
	format	[wipe] <i>path</i>	格式化一个 FAT 卷，可以选择在格式化之前，先把其中的数据擦除掉
storage	users		列出所有使用一个已 mount 的卷的进程的 PID（类似 fuser）
storage	mountall		调用 fs_mgr 去 mount fstab 中规定的所有文件系统
asec	list		列出所有 ASEC（Android Secure Storage，Android 安全存储）容器
	create	<i>cid mb fstype key uid</i>	创建一个新的 id 为 <i>cid</i> 、大小为 <i>mb</i> 、文件系统为 <i>fstype</i> 的 asec 容器
	destroy	<i>cid</i> [force]	删除 id 为 <i>cid</i> 的 asec 容器，可以用 force 参数，要求强制删除
	finalize	<i>cid</i>	finalize id 为 <i>cid</i> 的 asec 容器
	fixperms	<i>cid gid filename</i>	修改 id 为 <i>cid</i> 的 asec 容器中 filename 文件的访问权限，将它的拥有者改为 gid
	mount	<i>cid key uid</i>	用密钥 <i>key</i> mount app-id 为 <i>uid</i> 的 App 中 id 为 <i>cid</i> 的 asec 容器
	unmount	<i>cid</i> [force]	unmount id 为 <i>cid</i> 的 asec 容器。如果它当前正在使用，可以用 force 参数要求强制 unmount
	path	<i>cid</i>	返回 id 为 <i>cid</i> 的 asec 容器文件的存储路径
	rename	<i>old_cid new_cid</i>	将原来名为 <i>old_cid</i> 的 asec 容器重命名为 <i>new_cid</i>
	fspath	<i>cid</i>	返回 id 为 <i>cid</i> 的 asec 容器的 mount 点的路径



续表

命 令	子命令	参 数	作 用
obb	list		列出所有已经 mount 上来的 Opaque Binary Blob (OBB)
	mount	<i>filename key ownerGid</i>	用 App 的 ownerGid, 去 mount 由 filename 指定的 Opaque Binary Blob, key 是可选参数
	unmount	<i>source [force]</i>	unmount 由 source 参数指定的 Opaque Binary Blob
	path	<i>source</i>	
cryptfs	restart		向 init 发送信号, 要求重启一些框架
	cryptocomplete		验证文件系统是不是处于正常状态
	enablecrypto	<i>inplace wipe password</i>	加密文件系统, 可能会首先擦除掉该分区中已有的数据
	change pw	<i>default password pin pattern new_passwd</i>	修改加密密码
	checkpw	<i>passwd</i>	检查所提供的密码是不是正确, 能够 mount 加密的文件系统
	verifypw	<i>passwd</i>	供 BackupManagerService 使用
cryptfs	getfield	<i>name</i>	获取加密文件系统 (cryptfs) 中的元数据 (metadata) 字段
	setfield	<i>name value</i>	设置加密文件系统中的元数据字段
fstrim	do[d]trim[bench]		调用 FI[D]TRIM ioctl(2), 运行 mmc 驱动擦除未使用的块 (block)。在 M 版中新增了 bench (benchmark, 检测) 参数

从 L 版开始, Android 不再支持 xwarp 这个旧版 Android 中用在 YAFFS 文件系统上的命令了。在全面倒向 Ext4 之后, 这个命令已经过时了, 不过它仍然一直被保留到 KitKat 版——可惜, 已经没人再需要它了。

我们特别关心的是 vold 对加密文件系统的处理方式。在 Android 的官方文档<sup>[1]</sup>中, 详细解释了 Honeycomb 版中文件系统加/解密过程的实现细节。在接下来的部分中, 也将详细讨论这一部分。

mount 加密文件系统

从 Honeycomb 版开始, Android 引入了对磁盘加密的支持。通过扩展 Linux 中 dm-crypt 机制 (它同时还是 asec 机制的基础), Android 能够加密整个用户数据分区。系统分区仍是不加密

的，毕竟系统总是要以某种方式启动。不管怎么说，这都是个相当漂亮的设计——不论是在哪种设备上，系统分区都是一样的，而且实际上不会存储任何与用户相关的数据，所以加密系统分区基本没有什么好处。

我们将在第 8 章详细讨论 dm-crypt 特性，不过从比较高的视角看，我们现在只要知道 dm-crypt 就是透明地对块设备做加密和解密操作的就行了。不过在完成这些操作时，需要使用用户态程序提供加/解密口令。Android 可以在 /data 还没有被 mount 上来之前，通过用户解锁屏幕的操作获取解锁口令或解锁图形（如果用户已经启用了屏幕保护锁）<sup>1</sup>，并使用 com.android.settings.CryptKeeper 这个 activity，提示用户输入解锁设备所需的口令。

因此，这个流程实际上就是 init（控制系统的启动）、vold（提供实际的 mount 服务）和 CryptKeeper（负责显示给用户的 UI 界面）之间一串令人眼花缭乱的交互操作，如图 5-7 所示。

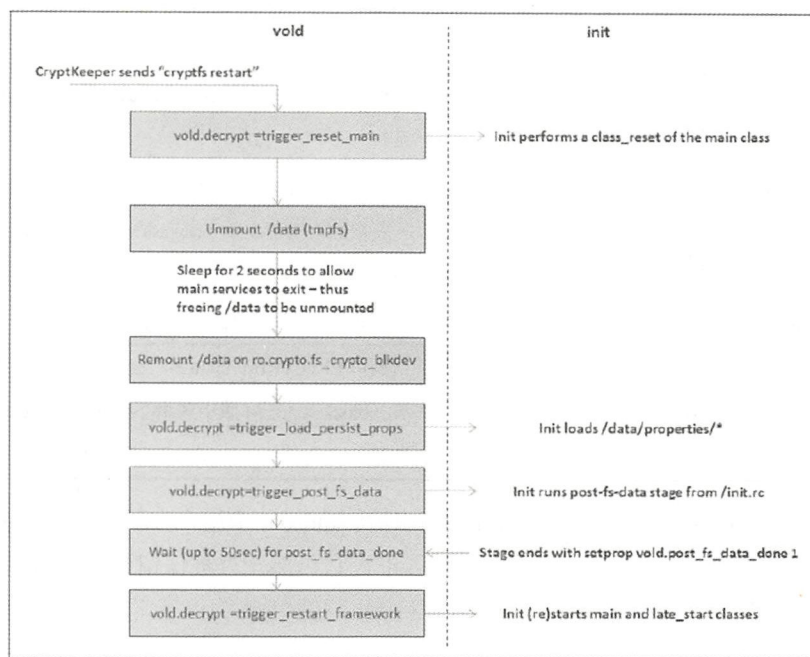


图 5-7 vold 和 init 之间的交互操作

在系统启动的过程中，init 调用 fs\_mgr 去 mount 文件系统，如果没有文件系统是加密的，它就会把系统属性 ro.crypto.state 设为“unencrypted”，然后去做“nonencrypted”这个 trigger 语句块中规定的动作——通常是启动被归在 late\_start 类中的服务。

1 Android Exporations blog<sup>[2]</sup>中有篇文章给出了如何使用 vdc cryptfs changepw 命令修改 Android 的加密口令。——原注



反之，如果有某个文件系统被加密了，那么显然在获得口令之前，加密文件系统是 mount 不上来的。这时，fs\_mgr 会用一个 tmpfs 代替这个文件系统，mount 在规定的地方，然后向 init 返回 1。于是 init 就会把系统属性 ro.crypto.state 设为“encrypted”，并通知 vold 需要解密/data 分区（具体方法是：把系统属性 vold.decrypto 设为 1）。在 KitKat 之前的 Android 版本里，vold 会使用系统属性 ro.crypto.tmpfs\_options 的值，作为 mount 的参数，但现在，这些参数已经被写在代码里了。mount（加密的）/data 分区首先要加载一个 UI 框架，而这又会需要向/data 分区写一些文件……解决这个死结的方法是先在/data 上 mount 一个 tmpfs（临时文件系统），而且由于这是个临时文件系统，所以现在写入的任何数据都不会被保留下来。在系统属性 vold.decrypto 被设为 1 时，SystemService 将只会运行属于“core”类的应用和服务。

com.android.Settings.CryptKeeper 把自己注册成系统的 home screen（使用 IntentFilter），并让自己拥有较高的优先级——因为它要确保自己能被首先启动。当它被加载的时候，它会用它的 onCreate() 函数中的代码，检查 vold.decrypto 的值。如果这个系统属性的值是 0 的话，它就直接退出，把位置让给“真正的”home screen。反之，如果确实有文件系统被加密了，CryptKeeper 就会在它的 onStart() 函数中启动一个异步的 ValidationTask，来连接 com.android.server.MountService，并调用它的 getEncryptionState() 函数，来检查该分区是不是确实被完全加密了。

前面说过，MountService 会连上 vold 的 socket。所以，它能向 vold 守护进程发送 cryptfs cryptocomplete 命令。这会让 vold 检查该加密分区是否确实处于正常状态（因为加密操作可能会突然被打断，导致/data 分区无法 unmount，并迫使用户重启手机或进入 recovery 状态）。如果 cryptocomplete 的返回结果表示文件系统正常，CryptKeeper 就会调用 SetupUi 要求用户输入解密用的口令或图形，然后再把得到的结果传递给 MountService，接着 MountService 又会通过 cryptfs checkpw 命令把它传递给 vold。

如果用户输入的口令是正确的，MountService 会发送 cryptfs restart 命令。这会让 vold 把系统属性 vold.decrypto 的值更新为 trigger\_reset\_main，并且休眠 2 秒钟，以便所有已加载了的属于 main 大类的服务能够全部停止，并把 tmpfs 从/data 这个位置 unmount 下来。如果 unmount 成功了，vold 就会把真正的（现在已经被解密出来了的）/data 分区重新 mount 上来。然后再次更新系统属性 vold.decrypt 的值，先是把它改成 load\_persist\_props<sup>1</sup>（因为这些 persist 的属性都是放在/data 分区里的），此后再把它改成 trigger\_post\_fs\_data（让 init 能够启动所有定义在 init.rc 中的必须使用/data 分区中文件的服务/操作），再接着又把它改成 trigger\_restart\_framework——这会使 init 重启相关的框架。上述这些属性值必须定义在/init.rc 文件中，与对应的 trigger 语句块相配套，如代码清单 5-9 所示。

---

1 原文如此，疑为 trigger\_load\_persist\_props，详见代码清单 5-9，或你手机里的 init.rc 文件。——译者注

代码清单 5-9 init.rc 中与加密事件相关的操作

```

on nonencrypted
    class_start late_start

on charger
    class_start charger

on property:vold.decrypt=trigger_reset_main
    class_reset main

on property:vold.decrypt=trigger_load_persist_props
    load_persist_props

on property:vold.decrypt=trigger_post_fs_data
    # This will call on post-fs-data, which must end with a post-fs-data-done
    trigger post-fs-data

on property:vold.decrypt=trigger_restart_min_framework
    class_start main

on property:vold.decrypt=trigger_restart_framework
    class_start main
    class_start late_start

on property:vold.decrypt=trigger_shutdown_framework
    class_reset late_start
    class_reset main

```

## 加密文件系统

加密文件系统的操作与加载加密的文件系统的操作十分类似。再重复一遍，UI 是由 CryptKeeper 生成的，并由 MountService 提供 Dalvik 虚拟机级到 vold 的桥接。UI 提示用户输入加密口令，并确认设备正在充电状态（这是为了防止加密操作进行到一半手机没电了……）。然后 MountManager 中的 encryptStorage() 方法会被调用，该方法会向 vold 发送 cryptfs enablecrypto 命令，这个命令的第一个参数可以是 wipe（在加密之前格式化/data 分区），也可以是 inplace，第二个参数是加密口令。

在收到该命令并验证它可以执行后，vold 会把系统属性 vold.decrypt 的值改成 trigger\_shutdown\_framework。这会导致 init 停止所有服务（“core”类中的服务除外），这实际上就是让系统回到了系统刚刚启动、用户还没有输入用户口令的状态。然后，/data 分区会被安全地 unmount 下来。vold 随后会把 vold.encrypt\_progress 设为初始值 0，并把系统属性 vold.decrypt 的值设为 trigger\_restart\_min\_framework，让 init 重新启动 main 类中的服务。

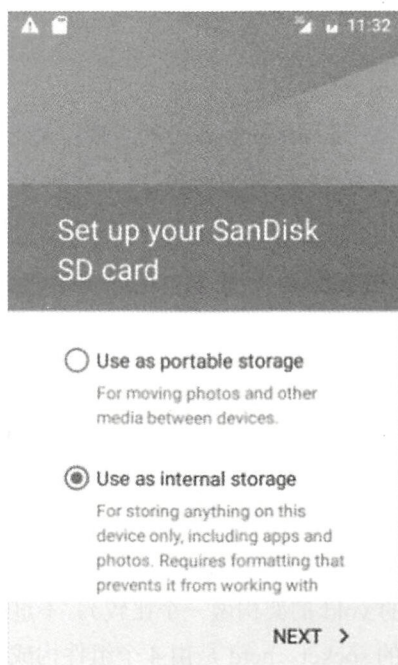
再说一遍，CryptKeeper 是作为一个全屏应用被加载的。它把读取到的 vold.encrypt\_progress 的值，显示在 UI 的进度状态条中。如果所有的操作都顺利完成了，进度条也就到 100% 了。如果有操作没能成功完成，vold.encrypt\_progress 的值就会被设为一个 error\_字符串。在 L 版中提供了“可恢复加密”（resumable encryption）这一功能，可是如果恢复失败了，用户就别无选择，

只能把设备重置为出厂状态了。

## Adopted Storage (Marshmallow)

从 Marshmallow 版开始, vold 通过 Adopting Storage, 新增了能够加密扩展存储器中的文件系统的功能。Adopted Storage 卷的加密方法和/data 的加密方法是一样的, 即使用 Linux 内核中的 dm-crypt 特性。其解密密钥(一个随机生成的 128 位密钥)由 vold 记录在/data/misc/vold 目录中。每个密钥都会被记录在一个单独的文件中, 这个文件的文件名是 `expand_GUID.key`, 其中斜体的 *GUID* 应该被替换成随机生成的分区识别 ID。因为(根据定义), 所有的 GUID 都是唯一的, 所以它们之间不会产生冲突。

Android 会弹出提示, 声明 Adopted Storage 卷只能在本机上使用(“use as Internal”), 它会被格式化成特殊的“防止它被用在其他设备上的”格式(如屏幕截图 5-1 所示)。不过话虽如此, 但如果你的手机已经 root 了, 你一样可以把解密密钥复制到另一台(已经 root 了的)手机里。这样, 这个 Adopted Storage 卷就能同时用在多台手机上了。



屏幕截图 5-1 格式化 Adopted Storage 卷时弹出的 UI 界面

真正格式化存储设备的工作是由另一个专门的二进制可执行文件/system/bin/sgdisk 来完成的, 所以 vold 需要通过命令行参数规定该程序的行为: `--new`(创建一个新的分区); `--change_name`



(指定卷的名称); --typecode= (指定 GUID 的类型); --partition-guid (指定卷的 GUID)。移动存储介质会使用 GPT 方案定义各个分区, 各个分区及其 GUID 如表 5-7 所示。

表 5-7 Adopted Storage 的分区和 GUID

Name	GUID(typecode)	用 途
android_meta	19A710A2-B3CA-11E4-B026-10604B889DCF	元数据分区——供今后使用, 每个物理存储设备会有一个这样的分区
android_expand	193D1EA4-B3CA-11E4-B075-10604B889DCF	加密分区

你可以使用新的 `sm upcall` 脚本去查询(`sm has-adopable`)或强行设置(`sm set-force-adoptable`) Adopted Storage 卷。在第 8 章中, 我们将从技术的视角, 以及内核的角度, 再来讨论文件系统的加密。

## 5.2 网络相关服务

### netd

Android 使用一个专门的守护进程来控制各个网络接口并管理它们的配置。如果你使用过 Android 中的 Tethering、防火墙或者 Wi-Fi 热点等特性, 甚至只要用过最基本的 DNS 查询功能, 你都能把自己视为 `netd` 的至尊用户。这个守护进程在 `/init.rc` 中的定义如代码清单 5-10 所示。

代码清单 5-10 netd 服务在 `/init.rc` 中的定义

```
service netd /system/bin/netd
class main
socket netd stream 0660 root system // Main interface for frameworks, and ndc
socket dnspoxyd stream 0660 root inet // Interface for Bionic, DNS resolution
socket mdns stream 0660 root system // Interface for NsdService, Neighbor Discovery
socket fwmarkd stream 0660 root inet // L: Firewall Marking interface
```

`netd` 和 `vold` 之间共享了许多相似的结构, 而且事实上也和 `vold` 共享了一部分代码——在它的 `socket` 处理代码中也使用了 `FrameworkListener` (及其他) 原生代码。图 5-8 显示了 `netd` 的架构 (你可以拿它和图 5-6 所示的 `vold` 的架构做一个比较)。不过和 `vold` 不一样的是, `netd` 中的每个子组件都会使用一个专门的 `socket`。`netd` 是由 4 个组件构成的, 下面我们一一讨论。

### CommandListener

`CommandListener` 负责监听 `/dev/socket/netd` 这个 `socket`, 以接收各个框架 (特别是我们下一章要讨论的 `NetworkManagementService`) 发来的命令, 并向连在该 `socket` 上的客户端发送 (广

播) 通知。和 `vold` 一样，在模拟器中含有一个简单的应用程序 `ndc`，这个应用程序可以被用作客户端，向 `netd` 发送命令并监听相关事件（使用 `ndc monitor`），如输出结果 5-9 所示。

```
root@htc_m8wl:/ # ndc monitor
[Connected to Netd]
600 Iface linkstate wlan0 up
614 Address updated 10.100.1.192/21 wlan0 128 0
614 Address updated fe80::522e:5cff:fef3:9da6/64 wlan0 128 253
```

输出结果 5-9 在连接一个无线网络时，使用 `ndc monitor` 命令得到的输出结果

在程序内部，`CommandListener` 把收到的命令分发给几个内部 `Controller` 类中的某一个，这些内部 `Controller` 类，各自负责某个特定方面的功能，其功能划分如表 5-8 所示。

表 5-8 netd 中的 Controller 类及其对应的命令

Controller	命 令	提供相应功能的程序	用 于
BandwidthController	bandwidth	/system/bin/ip[6]tables	网络配额（network quota）控制
ClatdController	clatd	/system/bin/clatd	464XLAT（通过 IPv6 网络中继的 IPv4 网络）控制
FirewallController	firewall	/system/bin/iptables	防火墙
IdletimerController	idletimer	/system/bin/ip, ip[6]tables	网卡空闲计时器
InterfaceController	interface	/system/bin/net/*	网卡
NatController	Nat	/system/bin/ip, ip[6]tables	网络地址转换（NAT）
PppController	ppp	/system/bin/pppd	VPN
SoftapController	softap	/system/bin/hostapd	WiFi-tethering/P2P
TetherController	Tether ipfwd	/system/bin/dnsmasq /proc/sys/net/ipv4/ip_forward	USB 和 WiFi-tethering

从图 5-8 中你可以看出（绝大部分的）`Controller` 并不提供任何功能的实现。与之相反，`Controller` 隐藏了额外的命令，通过调用 `fork()/exec(2)`，使用相应的守护进程或 `ip[6]tables` 完成相应的任务，实际上就是偷了个懒。我们把对 `Controller` 内部实现的完整讨论，以及它们调用的守护进程和框架留到第 2 本。表 5-9 让你能快速查阅到各个 `Controller` 使用的命令集。

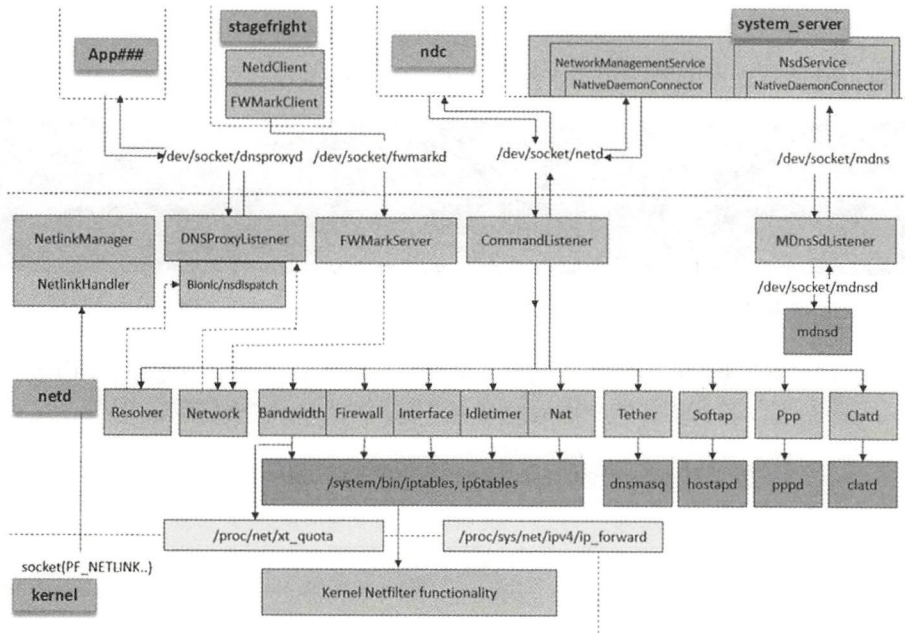


图 5-8 netd 中的 Controller 类及其对应的命令

表 5-9 netd 的 Controller 所能理解的 ndc 命令

Controller	子命令	用途
Bandwidth	enable	启用带宽配额控制
	setiquota iface qBytes	设置 iface 上的配额不超过 qBytes
	removeiquota iface	删除 iface 上之前设置的配额
	setifacealert iface qBytes	当 iface 上的带宽超过配额 qBytes 时，产生一个警告
	removeifacealert iface	删除之前 setifacealert 设置的带宽配额超出警告
	setglobalalert alertBytes	当任意一块网卡上的带宽超过配额 alertBytes 时，产生一个警告
	gettetherstats	获得设备 tethering 的使用情况
	setsharedalert   ssa bytes	设置在所有网卡上的总流量超过带宽配额 bytes 时，产生一个警告
	removesharedalert   rsa	
	addniceapps   aha uid	添加一个“Nice”（被允许的）App（用 uid 表示）
	removeniceapps   rha uid	
	addnaughtyapps   ana uid	添加一个“naughty”（行为不端的）App（用 uid 表示）



续表

Controller	子命令	用 途
Firewall	enable/disable	全局启用/禁用防火墙功能
	set_interface_rule <i>iface rule</i>	在网卡 <i>iface</i> 上应用指定的 iptables 规则 ( <i>rule</i> )
	set_egress_source_rule <i>addr rule</i>	设置基于源地址的发出流量过滤规则 ( <i>rule</i> )
	set_egress_dest_rule <i>addr port rule</i>	设置基于目的地址的发出流量过滤规则 ( <i>rule</i> )
	set_uid_rule <i>uid rule</i>	对指定的 <i>uid</i> , 应用一个 iptables 规则 ( <i>rule</i> )
IdleTimer	list	列出所有的网卡
	[en/dis]able <i>iface</i>	启用/禁用 Idletimer 机制: 启动并刷新 iptables 链
	[add/remove] <i>iface timeout classLabel</i>	在网卡 <i>iface</i> 上添加/删除一个 timer
Interface	list	列出所有的网卡
	route <i>iface default/secondary dest prefix gateway</i>	在路由表中添加一个项
	setmtu <i>iface mtu</i>	把网卡 <i>iface</i> 上的最大可转发包 (Maximum Transferrable Unit size) 大小设为 <i>mtu</i>
	ipv6 <i>iface enable disable</i>	在网卡 <i>iface</i> 上启用/禁用对 IPv6 的支持
	clearaddrs <i>iface</i>	删除在网卡 <i>iface</i> 上配置的 IP 地址
Interface	getcfg <i>iface</i>	显示网卡 <i>iface</i> 的配置
	setcfg <i>iface (ifconfig args)</i>	对网卡 <i>iface</i> 进行配置
	fwmark ..	防火墙标记 (Firewall marking) (从 L 版开始, 这一功能已经从 fwmarkd 中被删除了)
nat	[enable disable]int_ <i>iface</i> ext_ <i>iface</i>	启用/禁用网络地址转换 (NAT)
ppp	attach <i>tty local remote [dns1] [dns2]</i>	附加 PPPd 到 <i>tty</i> 上, 以建立一个 IP 地址 <i>local</i> 和 <i>remote</i> 之间的点对点连接。可以指定 DNS 解析服务器的 IP
	detach <i>tty</i>	执行/system/bin/pppd -detach
	list_ <i>ttys</i>	列出各个网卡所有被 ppp 守护进程使用的 <i>tty</i>
tether	start	启动伪 DNS 服务 (dnsmasq)
	stop	停止 tether: 使用一个 SIGTERM 信号, 杀掉伪 DNS 服务 (dnsmasq)
	dns [ <i>set list</i> ]	设置 DNS 或列出 DNS 设置
	interface [ <i>add remove</i> ] <i>iface</i>	在 <i>iface</i> 指定的网卡上添加/删除一个 tether

续表

controller	子命令	用 途
	ipfwd enable disable status	启用/禁用 IP 转发 (/proc/sys/net/ipv4/ip_forward)，或查询其状态
Resolver	setdefaultif iface	将 <i>iface</i> 指定的网卡用作默认的 DNS 查询网卡
	flushdefaultif	刷新默认网卡的 DNS 缓存
	flushif iface	刷新 <i>iface</i> 指定网卡的 DNS 缓存
	setifaceforpid <i>iface</i> <i>pid</i>	将 <i>iface</i> 指定的网卡分配给进程 ID 为 <i>pid</i> 的进程使用
	clearifaceforpid <i>pid</i>	取消将 <i>iface</i> 指定的网卡分配给进程 ID 为 <i>pid</i> 的进程使用这一设定
	Setifaceforuidrange <i>iface</i> <i>low</i> <i>high</i>	将 <i>iface</i> 指定的网卡分配给 <i>low-high</i> 指定范围内的 AID 使用
Softap	startap   stopap   status	(通过调用 exec()或 kill())执行/system/bin/hostapd)启用/禁用 Wi-Fi 热点，或查询其工作状态
	fwreload <i>iface</i> AP P2P STA	重新加载固件 (firmware)
	set <i>iface</i> SSID <i>hidden</i> /* <i>channel</i> security key	设置无线热点 (AP, access point) 的参数，除非其中有 “hidden”，否则 AP 将是可见的

DnsProxyListener

DnsProxyListener 是负责监听/dev/socket/dnsproxyd 这个专门用于名字解析(name resolution) 命令的 socket 的 FrameworkListener。相关命令如表 5-10 所示。

表 5-10 netd 的 DNS 代理命令 (DNS Proxying command) 子集

命 令	参 数	用 途
getaddrinfo	name service ai_flags ai_family ai_socktype ai_protocol <i>iface</i>	对 <i>iface</i> 指定的网卡,调用 getaddrinfo(3)。相比其他 getXXXbyYYY 函数, GAI <sup>1</sup> 是个更高级的向前兼容性更好的选择
gethostbyname	<i>iface</i> name af	执行一个正向 DNS 解析 (A/AAAA, 根据 af): 查找指定域名的 IP 地址
gethostbyaddr	addrStr addrLen addrFamily <i>iface</i>	执行一个反向 DNS 解析 (PTR): 查找指定 IP 地址对应的域名

Android 中的 LibC 实现 Bionic, 让所有的进程都能通过库调用表 5-10 中的命令, 并通过打开连到/dev/socket/dnsproxy 的 UNIX domain socket 的方法实现这些命令。这样, 所有的客户端 (不论它是原生代码还是跑在 Dalvik 虚拟机里的代码) 都可以通过 DNS 代理功能予以重定向。

1 GAI 就是 getaddrinfo(3)的缩写。——译者注

而且 `netd` 还可以根据发起调用进程的 `uid`，强制限定其 DNS 功能。因为每个 `uid` 都表示一个不同的应用，所以这也能在一个相当好的粒度上（根据每个不同的应用）控制 DNS 功能。

## mdnsd

组播 DNS（mDNS，Multicast DNS）是一个使用广泛的设备间相互发现协议，它最早是被苹果公司（作为“Bonjour”服务）所采用的。这个协议的标准是在 RFC6762 中确定的，它被广泛地用在 iOS 的“Air”协议族中（即“AirPlay”），允许设备通过向 224.0.0.253（或者 IPv6 的 FF02::fc）地址的 UDP 端口 5353 发送广播消息的方式相互发现。不出意料，Android 也选择采用这一标准，把它用作“WiFi Direct”的基础。从 JellyBean 版开始，`/init.rc` 增加了对 mDNS 服务的定义，见代码清单 5-11。

代码清单 5-11 `/init.rc` 中 mDNS 服务的定义

```
service mdnsd /system/bin/mdnsd
    class main
    user mdnsr
    group inet net_raw
    socket mdnsd stream 0660 mdnsr inet
    disabled
    oneshot
```

由于使用该协议必须要拥有能执行组播操作的权限，所以该服务既是 `inet` 组的成员（这使它拥有通用的 TCP/IP 权限），又是 `net_raw` 组的成员（这使它拥有诸如使用 raw socket，处理非标准的 IP 包的“高级”权限）。这个服务通过监听 `/dev/socket/mdnsd`（由 `MDNS_UDS_SERVERPATH` 这个宏定义）这个 socket 来获得相关请求，同时它还使用另一个连向 `netd` 的 socket——`/dev/socket/mdns`。

框架可以用 Network Service Discovery 类（`android.net.nsd`，从 API level 16 起引入）封装 mDNS 功能。这一点我们将在第 2 本书专门讲连接的那一章中详细讨论。mDNS 的实现本身（相关代码位于 `external/mdnsresponder` 目录中）与开源的 mDNS 项目是非常相似的，为配合 Android 特有的特性（比如 UNIX domain socket 和 Android 的日志）而做的相关修改，都被放在明确标注了“`#ifdef __ANDROID__`”的代码块中。

## mtpd

尽管服务名中的 `mtp`，在 Android 中（或者其他地方）一般都是和媒体传输协议（Media Transfer Protocol）联系在一起的。但 `mtpd` 和它的联系也仅仅到此为止了——这是个负责提供 PPP 和 L2TP（但不包括 IPSec）功能的守护进程。它在 `/init.rc` 中的定义如代码清单 5-12 所示。



代码清单 5-12 mtpd 服务在/init.rc 中的定义

```
service mtpd /system/bin/mtpd
class main
socket mtpd stream 600 system system
user vpn
group vpn net_admin inet net_raw
disabled
oneshot
```

该服务所在组所拥有的权能反映出：mtpd 服务需要能够访问网络（inet）、配置网卡（net\_admin）以及使用 IP 隧道（net\_raw）。作为一个被设为 oneshot 和 disable 的服务，mtpd 必须通过设置系统属性 ctl.start，手工来启动。事实上 com.android.server.connectivity.vpn（在它的 startLegacy 内部类中）就是这么干的。由于系统没有提供操作 VPN 功能的编程接口，所以该服务的启动或停止，只能通过 Android 系统的 GUI 来完成。

## racoon

racoon 已经成为了 VPN 守护进程的事实上的标准，作为一个外部项目，racoon 并不是 Android 的一部分，但它仍（在 Android 和 iOS 中）被广泛使用，以提供 VPN 服务（VPN 连接将在第 2 本书中予以讨论），见代码清单 5-13。

代码清单 5-13 mtpd 服务在/init.rc 中的定义<sup>1</sup>

```
service racoon /system/bin/racoon
class main
socket racoon stream 600 system system
# IKE uses UDP port 500. Racoon will setuid to vpn after binding the port.
group vpn net_admin inet
disabled # Started manually by ConnectivityManager
oneshot
```

注意：racoon 是以 root 身份启动的（这是为了能够绑定 ISAKMP 这个众所周知的特权端口），但随后它会降低自己的权限。这也就是为什么它还必须额外加入另几个组，以拥有建立网络连接的权能的原因（这些组的权限将在第 8 章予以讨论）。从严格的安全的角度讲，相对于拥有 root 权限，使用权能（特别是 CAP\_NET\_BIND\_SERVICE）实现相关功能是个更好的解决方案。特别是考虑到 racoon 中曾曝出过能被利用的漏洞的这段黑历史（事实上，它曾被用来对 iOS 5 系统进行越狱），这一改进就显得尤为重要。

---

1 原文如此，显然这里应该是“racoon 服务在/init.rc 中的定义”。——译者注

## rild

如果你的 Android 设备是一台手机或者带有 3G/LTE 上网功能的平板电脑，那么 rild 无疑就是非常重要的系统进程之一。事实上 RILD (**R**adio **I**nterface **L**ayer **D**aemon，无线接口层守护进程) 通过与基带 (baseband) 进行交互，在移动设备上提供了所有与电话相关的功能。该服务在 /init.rc 中的定义如代码清单 5-14 所示。

代码清单 5-14 rild 服务在 /init.rc 中的定义

```
service ril-daemon /system/bin/rild
    class main
    socket rild stream 660 root radio
    socket rild-debug stream 660 radio system
    user root
    group radio cache inet misc audio log
```

AOSP 提供的 rild 守护进程只不过是一个单纯的外壳而已：在解析了参数之后，它就会转而去调用厂商所提供的 RIL 库——这些库的位置可以由 -l 参数确定，也可以记录在系统属性 rild.libpath 中。由于这个库是被动态加载上来的，所以在加载时，会调用它的初始化函数——导出函数 RIL\_Init。rild 守护进程在继续运行，进入事件循环之前，这个初始化函数需要获取库中导出的各个 RIL 事件的处理函数，并一一注册它们。

尽管大多数用户认为这种方式早已过时了，但即使是在今天最新型号的手机中，所使用的与电话相关的 API，仍和上个世纪里调制解调器中使用的相差无几。事实上，底层的通话控制依靠的仍然是和调制解调器上使用的一模一样的命令集——“AT”命令集。如果你以前玩过小型机或者 Kermit<sup>1</sup> 的话，应该对它们相当的熟悉。rild 负责打开通话部件（它基本上就是一个串口），并产生上述这些命令。该守护进程也需要监听通话部分，以获取到那些“被动接收的命令” (unsolicited command)——这些命令通常是由基带产生的事件，比如有人打电话进来。

从图 5-9 和之前的定义可以看出：守护进程 rild 把 /dev/socket/rild 用作与应用交互的接口。手机中的应用可以通过这个 socket 连上这个守护进程，主动向基带发送各种与电话业务相关的请求 [比如：拨号 (dial)、接听 (answer)、挂断通话 (hang up)]。rild 也会使用这个 socket 向应用发出各种由基带产生的事件（比如收到短消息或有别的电话打进来）。但这并不是说应用能够直接使用这个 socket，这个 socket 会被封装成 Java RIL 的实现 (com.android.internal.telephony 包)。

1 一种由哥伦比亚大学设计的文件传输协议。——译者注

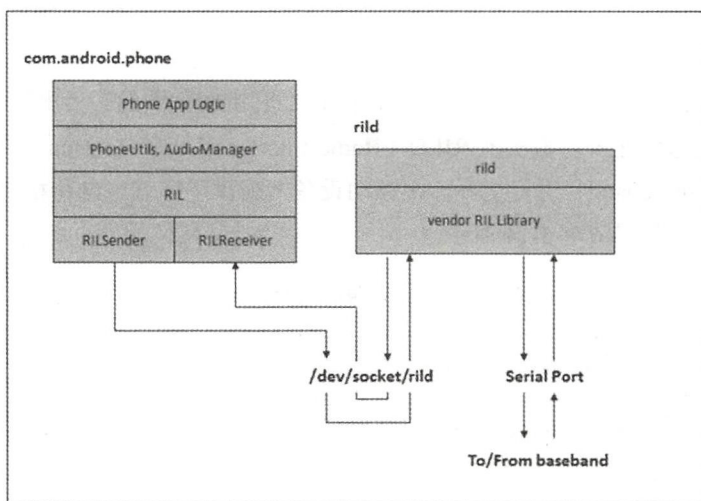


图 5-9 Radio Interface Layer 架构鸟瞰图

守护进程 `rild` 还会监听另一个 UNIX domain socket——`/dev/socket/rild-debug`。顾名思义，这个 socket 就是调试用的，它其实是个没有公开文档说明的、存储源自 `libril` 中的 `debugCallback` 的数据的地方。它定义了一系列你可以手工注入到 RIL 中，以模拟各种电话业务相关请求（用于调试）的代码。

此外，`rild` 还有一个它自己的调试工具 `radio`，它实际上是个专用的日志设备节点——`/dev/log/radio`。当你用“`logcat -b radio`”命令查看这个日志文件时，就会得到大量的调试信息，在某些版本的 Android 系统中，还会记下 `rild` 在拨号及建立连接过程中使用的“AT”命令。

在第 2 本中，我们将和 Java 技术框架一起，详细讨论 RIL（Radio Interface Layer，无线接口层），并说明 AOSP 中 RIL 相关部分的代码。

## 5.3 图形及多媒体服务

Android 中的图形和多媒体服务是系统尽可能提供最佳用户体验的不可分割的一部分。这一节中只能大致介绍一下这些服务，更详细的讨论要留到第 2 本深入讨论声音和图形内部实现机制时进行。

### surfaceflinger

`surfaceflinger` 居于 Android 图形栈的中心位置上。服务名中的“`flinger`”[有时也被称为“`compositor`”（排字工）]这个词是指将一个或多个层（Layer）的输入整合在一个层中予以统一



输出的角色。在 surfaceflinger 中，被整合的是图形“界面”（surface）（android.view.Surface 的各个实例），它既可以是框架提供的各种用户层输出的 view，也可以是开发者提供的 raw 或 GL Surface。在与 surfaceflinger 通信之前，框架会通过 servicemanager 寻找 surfaceflinger 服务，所以 surfaceflinger 并不需要使用 socket，这也使它在/init.rc 中的定义非常简单，如代码清单 5-15 所示。

代码清单 5-15 surfaceflinger 在/init.rc 中的定义

```
service surfaceflinger /system/bin/surfaceflinger
    class main
    user system
    group graphics
    onrestart restart zygote
```

尽管对 surfaceflinger 的深入讨论无疑是十分值得期待的（它将是第 2 本里专讲图形的那一章中的重点），但我们还是可以通过图 5-10（尽管该图已经经过了一些简化），从全局的高度，对 surfaceflinger 在 Android 图形架构中所处的位置有个大致的概念。

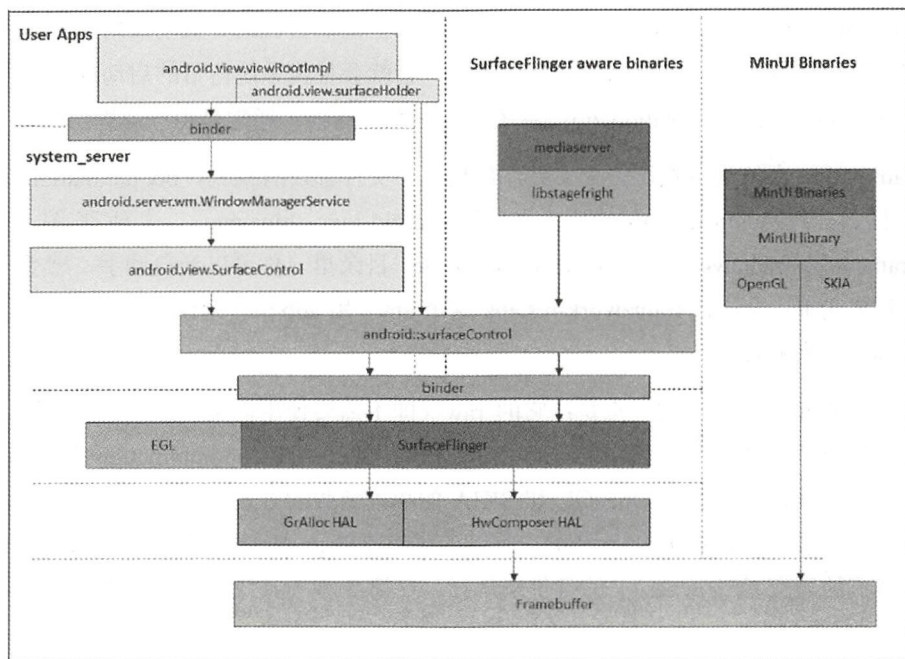


图 5-10 surfaceflinger 功能的鸟瞰图

## bootanimation

bootanimation 这个服务是/system/bin 目录下一个很小的二进制可执行文件，它专门被 surfaceflinger 用作在它〔和其他多媒体（media）框架〕被加载时的占位符。因此，它在/init.rc 中的定义如代码清单 5-16 所示。

代码清单 5-16 bootanimation 在/init.rc 中的定义

```
service bootanim /system/bin/bootanimation
    class main
    user graphics
    group graphics
    disabled      # started by SurfaceFlinger using ctl.start
    oneshot
```

这个二进制可执行文件实质上是个很简单的程序——它启动之后，只是去寻找 3 个 zip 包。

- /system/media/bootanimation-encrypted.zip: 当系统属性 vold.decrypt 被设为 1（这表示文件系统已经被加密了）时，需要使用这个压缩包。
- /data/local/bootanimation.zip:（高级）用户可以把他们自己的开机动画用 adb push 命令上传到这个位置上，如果这个文件存在的话，它就会替代系统自带的启动动画。
- /system/media/bootanimation.zip: 系统默认的开机动画，通常是由厂商提供的。

bootanimation 会依次寻找这三个文件，如果三个文件都没有找到，bootanimation 默认会交替显示两张图片 android-logo-mask.png 和 android-logo-shine.png，这两张图片都藏在 /system/framework/framework-res.apk 的/assets/images 目录里（你可以亲自动手，感受下挖出这两张图片有多简单：只要把 framework-res.apk 这个 apk，用 adb pull 命令复制到你的电脑上，然后再把它解压出来就行了）。

bootanimation 最有意思的一点是：它的 raw（即不需要使用框架）图形权能。因为它是最先被启动的几个服务之一，那个时候框架还没有初始化呢！所以 bootanimation 只能自己动手，丰衣足食，以直接调用底层的 OpenGL 和 SKIA 的相关函数的方式自给自足。正是因为需要直接访问设备的帧缓存（/dev/graphics/fb0），所以它才需要以 graphics 这个 uid〔（/dev/graphics/fb0 这个设备节点的所有者（owner）〕的身份来运行。在第 2 本中，我们还将更进一步地讨论与底层图形相关的函数。



因为使用底层调用，直接对帧缓存进行写操作，这也让 bootanimation 能够在 surfaceflinger 已经运行了的情况下，完全覆盖 surfaceflinger 的屏幕输出。你可以用 adb shell 在你的手机上运行 bootanimation 的方式，来验证这一点。你的手机上或许默认已经有一个 bootanimation 了，但你也

以把从模拟器镜像中获取到的 `bootanimation` 上传到手机上。

在你的手机正在运行时运行 `bootanimation`，会使手机屏幕上出现开机动画，覆盖原来屏幕上的内容——屏幕可能会被全部覆盖（在竖屏模式下），也有可能只是被覆盖一部分（在横屏模式下）<sup>1</sup>。这时，你对屏幕的触摸操作仍然是有效的，只不过在你退出 `bootanimation`（按下 `CTRL-C` 组合键退出该程序）之前，你很可能根本看不到你到底在干些什么。

大多数厂商会提供它们自己的 `bootanimation.zip`，在里面装上它们自己的 Logo（在有些情况下会使用电信运营商的 Logo）。而 `cyanogen` 及其他一些 Android ROM 刷机包里也会给出刷机团队自己的 `zip` 包，作为开机动画。这些 `zip` 包中必须含有一个 `desc.txt` 文件，以及许多张 `bootanimation` 会不断循环展现的图片。第一帧会与 ROM 的启动图画（如果有的话）重叠，以确保动画的平滑过渡。

注意，有些厂商会替换默认的二进制可执行文件，以使用它们自己的开机动画及配套的音乐（比如三星就是用 `/system/bin/samsungani` 和 `qmg` 属性中指定的文件这么干的）。此外，厂商也可能会修改 `bootanimation` 搜索开机动画的路径（比如 HTC One M8 就会到 `/system/customize/resource` 目录中去搜索 `hTC_bootup_one.zip` 这个文件）。Kindle 中使用的“FireOS”的做法介于这两者之间，它保留了 `bootanimation` 这个二进制可执行文件，但对它进行了修改，以显示“Kindle Fire”的 Logo 而不是 Android 的 Logo。

本书的官网<sup>[3]</sup>上提供了不少开机动画，你可以用它们在自己的手机上做些实验。你可以把它们放到手机的 `/data/local` 目录中（因为这个目录会排在 `/system/media` 目录之前被搜索，这样你能看到替换开机动画有多么简单）。你必须确保 `zip` 文件是可读的（否则，当你在 `adb` 中用普通用户的 `shell` 运行 `bootanimation` 时，开机动画就会变成默认的字符串“`ANDROID`”）。

### 实验：查看 `bootanimation` 所使用的 `zip` 文件的结构

在 Nexus 5 这类（谷歌自己的）手机中，开机动画就放在 `/system/media/bootanimation.zip` 文件中，你可以使用 `adb pull` 命令把它下载到电脑上，像输出结果 5-10 所示的那样，查看其中的内容。

`desc.txt` 文件的第一行中依次规定的是开机动画的宽度、高度以及每秒显示的帧数（`fps`）。注意，组成开机动画的各个 `png` 图片也必须严格符合这一规格。

不过，在其他手机/平板电脑中，要想知道开机动画到底是放在哪个文件里的，还需要有那么点逆向功底，幸运的是，在 `strace` 的帮助下，即使是最复杂的 `bootanimation` 也会招出它的秘

1 原文如此，译者在一台 Nexus S 手机（安装的是 Android 4.1）上实验的结果是：无论是横屏还是竖屏，开机动画都会覆盖整个屏幕，只有在横屏/竖屏切换后，才会出现屏幕被部分覆盖的情况。——译者注



密。比如像输出结果 5-11 所演示的，就是在 HTC One H8 手机中，用 strace 探析开机动画存放位置的方法。

```
morpheus@Forge (~)$ adb pull /system/media/bootanimation.zip
1275 KB/s (1068873 bytes in 0.818s)
morpheus@Forge (~)$ unzip -t bootanimation.zip
Archive: bootanimation.zip
  testing: desc.txt                OK
  testing: part0/                  OK
  testing: part0/000.png           OK
  testing: part0/001.png           OK
  ..
  testing: part1/059.png           OK
# After unzipping:
morpheus@Forge (~)$ cat desc.txt
1080 230 24
p 1 0 part0
p 0 0 part1
morpheus@Forge (~)$ file part1/000.png
000.png: PNG image data, 1080 x 230, 8-bit/color RGB, non-interlaced
```

输出结果 5-10 bootanimation.zip 样例

```
# Using strace: -f: to follow forks and threads (since binary may be multi threaded)
#               -o: to save output to a local file
#
shell@htc_m8wl:/ $ strace -f -o /data/local/tmp/out /system/bin/bootanimation
#
# Let the bootanimation run, watch logo, then hit CTRL-C.. and sift through output
#
shell@htc_m8wl:/ $ grep open /data/local/tmp/out |
| grep -v /dev | grep -v /proc | grep -v /lib
...
21217 open("/system/etc/customer/bootanimation.zip", O_RDONLY) = -1 ENOENT
21217 writev(4, [{"\5", 1}, {"zipro\0", 6}, {"Unable to open '/system/etc/"..., 88}], 3) = 95
21217 open("/data/data/com.htc.CustomizationSetup/files/boot_anim_mns", O_RDONLY) = -1 ENOENT
21217 open("/system/customize/CID/default.xml", O_RDONLY) = 10
21217 open("/system/customize/resource/vzw_bootup.zip", O_RDONLY) = 10
...
```

输出结果 5-11 探究 HTC One H8 手机的开机动画是放在哪里的

如输出结果 5-11 所示，HTC One H8 手机为每个运营商都定制了开机动画（在这个例子里使用的是一台 Verizon 手机）。替换掉/system/customize/resource/vzw\_bootup.zip，或者添加/system/etc/customer/bootanimation.zip（因为这是首先会被搜索的位置），就能改掉开机动画。

## mediaserver

mediaserver 是 Android 最重要的组件之一。这个服务是多媒体处理的中心节点，控制着录-播功能。它在/init.rc 中的定义如代码清单 5-17 所示。

代码清单 5-17 mediaserver 在/init.rc 中的定义

```
service media /system/bin/mediaserver
class main
user media
group audio camera inet net_bt net_bt_admin net_bw_acct drmrpc mediadrn
ioprio rt 4
```

从这个服务所属的用户组，我们可以看出，mediaserver 需要访问声音、摄像头、网络服务以及 DRM 框架（稍后会讨论它）的权限。不过，mediaserver 实际上只是各个真正的服务的容器，这一点类似于 Windows 中的 service host (svchost.exe) 的概念。表 5-11 中列出了其中容纳的各个服务。

表 5-11 mediaserver 中容纳的各个服务

服 务	公开的名称	提供的服务
AudioFlinger	media.audio_flinger	音频播放，该服务获取一个或多个 PCM 音频流作为输入，然后把它们“合并”到一个混合流中
AudioPolicyService	media.audio_policy	音频策略，告诉 AudioFlinger 音量大小以及目标音频播放设备
CameraService	media.camera	摄像头服务，它是“相机”App（不论它是 Android 中自带的“camera”App 还是厂商提供的相机 App）的主要客户端
MediaPlayerService	media.player	播放视频和音频

KiKat 版（及以后的版本）中的 mediaserver 通过提供 registerExtensions()函数，支持对服务进行扩展——尽管目前还没有定义什么扩展服务。我们将在第 2 本更详细地讨论 mediaserver 中容纳的各个服务。

实验：通过 media.log 服务调试 mediaserver

mediaserver 有一个很有用的调试特性，那就是：在启动 mediaserver 服务时，系统会去检查系统属性 ro.test\_harness 的值。如果这个值是 1 的话，系统就会把 mediaserver fork()成 MediaLogService 进程（media.log）的子进程，这样 media.log 就能（利用父进程的权力）获取到 mediaserver 使用资源的实时状态信息了。这可以从输出结果 5-12 这个添加了注释的输出结果中看到。

```
# Make sure property check exists in the binary (note: Crude, may yield false positive)
root@htc_m8wl:/ # grep ro.test_harness /system/bin/mediaserver
Binary file /system/bin/mediaserver matches
# Set the property
root@htc_m8wl:/ # setprop ro.test_harness 1
# Kill the media server
root@htc_m8wl:/ # kill -9 $(mediaserver pid)
# Et voila! media.log is now the parent of mediaserver
root@htc_m8wl:/ # ps | grep media
media      19122 1      20548  6444  ffffffff b6edaab0 S media.log
media      19123 19122 59876  9520  ffffffff b6edb26c S /system/bin/mediaserver
root@htc_m8wl:/ # service list | grep media.log
0          media.log: [android.media.IMediaLogService]
```

输出结果 5-12 使用 ro.test\_harness 启动 media.log 服务

这样做了之后，你就可以在 mediaserver 的整个生命周期中用 logcat |grep media.log 或 dumsys media.log 命令查看它的资源使用情况了。

## drmserver

Android 为有防复制保护需求的内容提供了一个数字版权管理（DRM，Digital Rights Managerment）框架，而 drmserver 则是被用作所有 DRM 请求的中心节点的组件。它在/init.rc 中的定义如代码清单 5-18 所示。

代码清单 5-18 drmserver 在/init.rc 中的定义

```
service drm /system/bin/drmserver
    class main
    user drm
    group drm system inet drmrpc
```

事实上，Android 提供的这个所谓的“框架”多少有些名不副实，因为它只定义了一些 API，根本就没有任何真正的内容验证逻辑。实际工作是通过一个插件架构留给厂商来实现的。而这些插件实际上就是一些的动态链接库（so 文件，即 shared object）文件，它们被存放在 /vendor/lib/drm 和/system/lib.drm 目录中，drmserver 会逐个加载这两个目录中的 so 文件。drmserver 也因此变得非常小，它仅仅是由 main() 函数中的寥寥数行代码组成的，这些代码会在 ServiceManager 那里注册一个 DrmManagerService (drm.drmManager)，并在有来自框架的 DRM API 调用出现时，调用其内部的 DrmManager 类，找到与该 DRM 内容对应的插件，然后就把球踢给这个插件去处理。



```

shell@android:/ $ ls -l /vendor/lib/drm
/vendor/lib/drm: No such file or directory # No vendor specific DRM modules
1|shell@android:/ $ ls -l /system/lib/drm
-rw-r--r-- root      root      48336 2012-05-2
-rw-r--r-- root      root     117224 2012-05-21
-rw-r--r-- root      root      68944 2012-05-
-rw-r--r-- root      root      48604 2012-05-21 1
-rw-r--r-- root      root      65312 2012-05-
-rw-r--r-- root      root      48212 2012-05-2
-rw-r--r-- root      root      65012 2012-05-21 17:
-rw-r--r-- root      root      64836 2012-05-2

```

输出结果 5-13 查看 Galaxy S3 中的 DRM 插件

为了能被 `DrmManager` 视为一个有效的插件，并能被其调用，插件编写时必须遵循定义在 `IDrmEngine.h` 中的 `IDrmEngine` 接口标准。关于该标准更详细的讨论（包括使用一个 `PassThru` 模块检查 DRM 消息流）将在第 2 本中进行。

## 5.4 其他服务

“main”大类中剩下的这些服务都很难被归为一类——因为它们分别体现了系统支持的各个不同的方面，但这并不会在任何程度上减弱它们重要性。

### installd

`installd` 这个守护进程是负责安装或卸载 App 包（package）的。不论你的 App 包是如何安装的 [不论是直接从谷歌市场（Google Play）下载安装的，还是用 `adb install` 安装的]，最终都还是要调用 `installd` 的。不过这个守护进程本身，还是被动式的：它会去监听一个由 `init` 安装的 socket，Android 框架产生的命令将由这个 socket 传递给它。在这个守护进程的 `/init.rc` 中定义了这个 socket，如代码清单 5-19 所示。

代码清单 5-19 `installd` 在 `/init.rc` 中的定义

```

service installd /system/bin/installd
class main
socket installd stream 600 system system

```

### 启动过程

`installd` 的启动过程如图 5-11 所示。

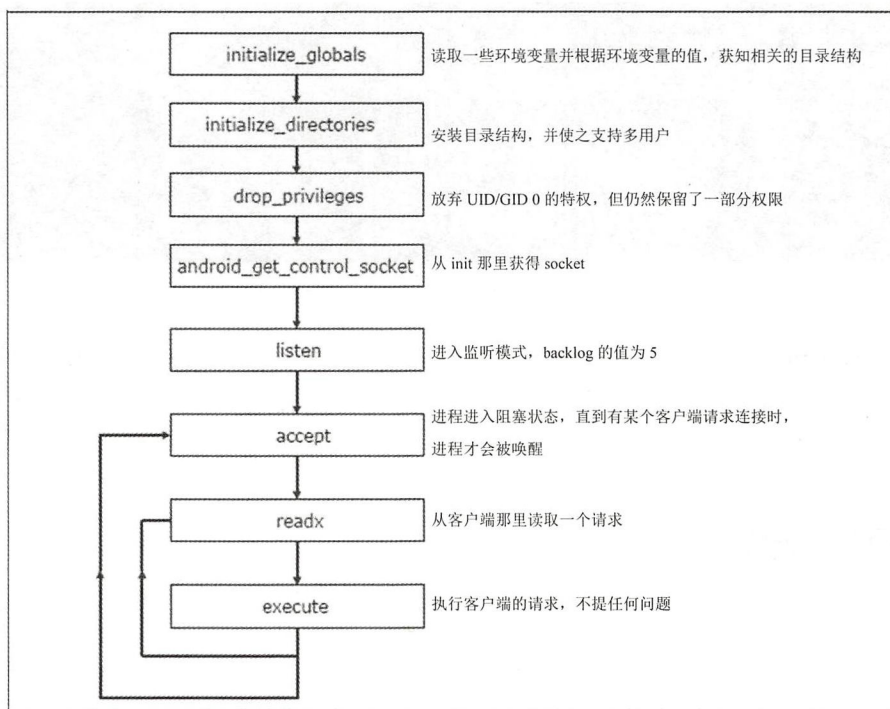


图 5-11 installd 的启动过程

在启动时，`installd` 需要安装和维护一个目录结构——各个 App 都会被安装到这个目录结构中。这个目录的根目录（base name）是记录在 `ANDROID_DATA` 环境变量中的。`installd` 还会在这个根目录后面添上 `APP_SUBDIR` 环境变量（`app/`）、`PRIVATE_APP_SUBDIR` 环境变量（`app-private/`）、`APP_LIB_SUBDIR` 环境变量（`app-lib`）和 `MEDIA_SUBDIR` 环境变量（`media/`）的值。

`Installd` 还会再读取另外两个环境变量 `ANDROID_ROOT`（指向 `/system`）和 `ASEC_MOUNTPOINT`（指向 `/data/asec`）。在得到了该目录结构之后，`installd` 会去初始化这些目录，以确保它们是存在的（因为在默认的出厂设置中，`/data` 中已经建立了相关目录，只是其中没有任何内容罢了）。至少从 JellyBean 版开始，`installd` 还要负责修改该目录结构，使之支持多用户的任务，具体操作步骤如下：

- 创建 `/data/user` 目录，该目录的拥有者是 `system:system`，访问权限是 `rw-x-x-x`。
- 创建一个符号链接 `/data/user/0` 使之指向 `/data/data`。
- 把 `/data/media` 升级为 `/data/media/0`，并把之前存放在 `/data/media` 中的所有文件移到 `/data/media/0` 中。

- 为系统中的其他所有用户创建/data/media/##目录。
- 把/data/media/0/Android/obb 中的各个 OBB 文件移到/data/media/obb 目录中，这样，它们才能在各个用户之间共享，并减少在文件系统中占用的空间。
- 确保各个用户的 media 目录存在 (/data/media/##) 以及这些目录的拥有者和访问权限为 media:media rwxrwx---。

尽管 `installd` 是以 `root` 身份启动的，但至少从 Jellybean 版开始，`installd` 开始应用最小权限原则。为此，它首先要去调用的函数之一就是 `drop_privileges()`，该函数把这个进程的 `uid/gid` 设为 `AID_INSTALL`。它也会使用 Linux 权能（详见第 8 章）以维护 `chown(2)`、`setuid(2)`/`setgid(2)` 和 `DAC`，因为在安装或卸载由不同的用户或组 `id` 拥有的各种包（`package`）时，它需要这些权限。

最后，`installd` 会在获得一个控制 `socket (/dev/socket/installd)` 后，进入一个等待获得消息的循环。当有某个客户端连上了这个 `socket` 时，`installd` 就会进入一个内部的读取/执行循环，来处理这个连接，直到该连接关闭为止（这也就意味着，在任意给定的时刻，`installd` 只能对一个客户端进行处理）。我观察到一个有意思的情况是：`installd` 从不对 `socket` 中的“`call id`”进行任何验证，而是靠该 `socket` 已经被 `chmod()` 为 `system:system rw-----` 权限，来保证安全性。因为同一时刻只能处理一个客户端，所以可以默认假设它是 `PackageManager` 所有的。另外也应该注意到，`installd` 也不会对 `APK` 的签名进行验证——它假定它的调用者已经完成了这一步验证。

相关命令

框架可以使用尚无文档说明的 `com.android.server.pm.Installer` 类来使用 `PackageManager`，这个类提供了一个 Dalvik 虚拟机级的 API，受信的应用可以用它来安装或卸载各种 App。API 中的各个方法都会被映射为通过 `socket` 发送的（以表示长度的 2 个字节的字符串的前缀形式）命令，相关命令如表 5-12 所示。

表 5-12 `installd` 使用的相关命令（加了灰底的是 L 版中新增的命令）

命 令	参 数	用 途
ping		空命令，用来检查连接状态
install	<i>pkgname uid gid seinfo</i>	安装由 <i>pkgname</i> 参数指定的包（ <code>package</code> ），这个包的所有者/组由 <i>uid/gid</i> 参数指定，SELinux 上下文由 <i>seinfo</i> 参数指定
dexopt	<i>pkg_path uid is_public</i>	优化 APK 中的 dex 文件，这一操作会产生一个 <code>.odex</code> 文件
movedex	<i>src dst</i>	把 <i>src</i> 参数指定的 DEX 文件重命名为 <i>dst</i>
rmdex	<i>pkg</i>	删除由 <i>pkg</i> 参数指定的包中的 DEX 文件
remove	<i>pkgname uid</i>	删除在 <i>uid</i> 身份下安装的 <i>pkgname</i> 包（ <code>package</code> ）



续表

命 令	参 数	用 途
Rename	<i>oldname newname</i>	把 <i>oldname</i> 参数指定的包重命名为 <i>newname</i>
fixuid	<i>pkgname uid gid</i>	把 <i>pkgname</i> 参数指定的包的拥有者/组改成 <i>uid</i> 和 <i>gid</i>
freecache	<i>free_size</i>	释放 cache 空间，直到有了 <i>free_size</i> 字节的空闲空间为止
rmcodecache	<i>pkgname uid</i>	从 cache 中删除拥有者为 <i>uid</i> 的 <i>pkgname</i> 包的代码缓存
rmcache	<i>pkgname uid</i>	从 cache 中删除拥有者为 <i>uid</i> 的包 <i>pkgname</i>
getsize	<i>pkgdir uid apkpath</i>	返回由 <i>apkpath</i> 参数指定的目录所占的空间大小
rmuserdata	<i>pkgname uid</i>	删除拥有者为 <i>uid</i> 的包 <i>pkgname</i> 使用的用户数据
movefiles		执行/system/etc/updatecmds 中的脚本
linklib	<i>pkgname asecLibDir uid</i>	链接原生库（native library）到它的真实存储位置
mkuserdata	<i>pkgname uid userid</i>	为 <i>pkgname</i> 包创建数据目录（拥有者是用户 <i>userid</i> 的 id），并安装符号链接
mkuserconfig	<i>uid</i>	确认/data/misc/user/ <i>uid</i> 目录存在
rmuser	<i>uid</i>	删除 <i>uid</i> 参数指定的用户
idmap	target overlay	运行/system/bin/idmap
restorecondata	<i>pkgname seinfo uid</i>	恢复拥有者为 <i>uid</i> 的包 <i>pkgname</i> 的 <i>seinfo</i>
patchoat	<i>apk_path, uid, is_public, package, instruction_set, vm_safe_mode, should_relocate</i>	修补 OTA 文件，并把它重新加载到内存中

包的安装过程以及 Installer 服务，将在第 2 本书中一并讨论。

keystore

顾名思义，keystore 服务提供的就是密钥（key）存储服务。尽管其设计初衷是提供能够存储任意给定的名称-值（name-value）对，但在实践中，它只被用来存储密钥。该服务在/init.rc 中的定义如代码清单 5-20 所示。

代码清单 5-20 keystore 在/init.rc 中的定义

```
service keystore /system/bin/keystore /data/misc/keystore
class main
user keystore
group keystore drmrpc
```

传给守护进程 keystore 的参数/data/misc/keystore 是用来存储各个 keystore 文件的目录的。从 JellyBean 版开始，每个用户都会有他自己的 keystore 目录，第一个用户（primary users）使

用/data/misc/keystore/0 这个目录。用户的 keystore 的口令被存放在一个 .masterkey 文件中（根据用户的屏幕解锁密码推出的密钥予以加密），各个 App 的 keystore 则存放在名为 AID\_XXXX 的各个对应的文件中。

从 4.4 版起，keystore 就不再使用 socket 了——这一点和其他的守护进程不太一样。keystore 只能通过 servicemanager，使用名称 android.security.keystore 来访问。keystore 服务的框架（framework）客户端是 java.security.KeyStore 类，这是一个依照 Java 标准构造的，可供开发者访问的类，因此也理所当然地有文档支持<sup>[4]</sup>（尽管文档并不完备：有相当一部分的 public 的方法，被 Android 的文档有意忽略掉了）。keystore\_cli 这个命令行工具让我们能部分地在命令行中通过原生代码访问 keystore。表 5-13 中给出了 Java 类和服务接受的命令，我使用灰底表示那些并没有在 cli 中实现的命令。

表 5-13 keystore 使用的命令，由 java.security.keystore 导出

1	test()	测试 keystore 守护进程是否是活跃的
2	byte [] get(String name)	获取与指定名称 <i>name</i> 对应的值
3	insert(String name, byte [] val, int uid, int flags)	向属于 <i>uid</i> 的 keystore 插入一个名称-值对，flags 由参数 <i>flags</i> 指定
4	int del(String name, int uid)	从属于 <i>uid</i> 的 keystore 中删除名称 <i>name</i> （及其对应的值）
5	exist(String name, int uid)	检查在属于 <i>uid</i> 的 keystore 中是否存在名称 <i>name</i>
6	saw(String prefix, int uid)	列出属于 <i>uid</i> 的 keystore 中所有以前缀 <i>prefix</i> 开头的键（key）
7	reset()	重置（擦除）keystore 中的内容
8	password(String password)	把 keystore 的口令修改为 password
9	lock()	锁定 keystore，需要提供口令才能解锁之
10	unlock(String password)	提供口令，解锁之前锁定的 keystore
11	zero()	检查 keystore 是否为空
12	generate(String name, int uid, int keyType, int keySize, int flags, byte [] [] args)	产生一对公钥/私钥，并将它们存储在属于 <i>uid</i> 的 keystore 中的名为 <i>name</i> 的名称-值对中。被存储下来的数据被称为“密钥”（key），该密钥（key）可以用来对数据进行签名、验证，或在保持私钥不可访问的前提下，提取出公钥
13	import_key(String name, byte [] data, int uid, int flags)	把属于 <i>uid</i> 的 keystore 中的名为 <i>name</i> 的名称-值对中存储的密钥（key），导出到名为 <i>data</i> 的二进制数据块中
14	byte [] sign(String name, byte [] data)	用 <i>name</i> 中存储的密钥（key）对数据 <i>data</i> 进行签名，但并不真正提取出 key

续表

15	verify(String name, byte[] data, byte [] signature)	用 <i>name</i> 中存储的密钥（key）验证数据 <i>data</i> 的签名
16	byte [] get_pubkey(String name)	获取与给定的名称 <i>name</i> 相关联的公钥
17	del_key(String name, int uid)	在属于 <i>uid</i> 的 keystore 中删除指定名称 <i>name</i> 对应的 key
18	grant(String name,int uid)	授权 <i>uid</i> 可以访问名为 <i>name</i> 的密钥
19	ungrant(String name, int uid)	撤销对 <i>uid</i> 可以访问名为 <i>name</i> 的密钥的授权
20	long getmtime(String name)	获取给定的名称 <i>name</i> 的最后一次修改时间
21	duplicate(String srcKey, int srcUid, String destKey, int destUid)	将属于 <i>srcUid</i> 的 keystore 中 <i>srcKey</i> 名称-值对中存储的密钥(key), 复制到属于 <i>destUid</i> 的 keystore 中的 <i>destKey</i> 名称-值对中
22	is_hardware_backed(String keyType)	返回一个整型数，表示 <i>keyType</i> 是否已经由一个硬件 keystore 备份了
23	clear_uid(long uid)	清空 id 为 <i>uid</i> 的用户的 keystore
24	reset_uid(long uid)	重置 id 为 <i>uid</i> 的用户的 keystore
25	sync_uid(long uid)	同步 id 为 <i>uid</i> 的用户的 keystore
26	password_uid(long uid)	设置 id 为 <i>uid</i> 的用户的 keystore 的口令

（命令执行的）返回码是定义在 system/security/keystore/include/keystore/keystore.h 文件中的，这些返回码会被 keystore\_cli 翻译成如表 5-14 所示的字符串。


表 5-14 keystore 的出错码

出错码	对应的常量
1	[STATE_]NO_ERROR
2	[STATE_]LOCKED
3	[STATE_]UNINITIALIZED
4	SYSTEM_ERROR
5	PROTOCOL_ERROR
6	PERMISSION_DENIED
7	KEY_NOT_FOUND
8	VALUE_CORRUPTED
10-13	WRONG_PASSWORD_[0123]
14	SIGNATURE_INVALID



密钥的访问是由 uid 来管理的（所以它也被用作一个参数），因此实际上每个应用都有它自己的私有存储。另外，通过在 init 中写死一些类似于 ACL 的东西，keystore 守护进程维护了一个名为 user\_perms 的权限数组，其中特别为 AID\_SYSTEM（全局皆可访问）、AID\_VPN 和 AID\_WIFI（仅在 get、sign 和 verify 时）指定了一些例外。事实上，AID\_ROOT 用户是最受限的——只允许进行 get 操作（尽管在实际操作中，把它 su 到 AID\_SYSTEM 也不过是举手之劳）。

### 实验：keystore 的接口

 下面这个实验（如果没能正确完成相关操作的话）可能使你无法再访问你的 keystore，因此建议你在模拟器镜像上，而不是一台实体手机/平板电脑上做这个实验——除非你有足够的自信。

你既可以使用 keystore\_cli 命令行程序，也可以直接使用 service call 的方式调用 keystore 服务。不过考虑到（至少截至本书编写之时）并非所有的命令都已经在 keystore\_cli 中实现了，所以直接使用 service call android.security.keystore 并传给它用数字形式表示的功能码（code），再加上相关参数的方式，能够实现的功能会更多一些。输出结果 5-14 中以“分屏截图”的方式，将这两种不同的调用 keystore 的方法，一一对应地展示了出来。

```
# Note root is not allowed access to the keystore
...# keystore_cli test                                service call android.security.keystore 1
test: Permission denied (6)                               Result: Parcel(00000000 00000006  '.....')
#
# SU to system to gain full control
...# su system
...$ keystore_cli test                                service call android.security.keystore 1
test: No error (1)                                       Result: Parcel(00000000 00000001  '.....')
#
# Set keystore password to be "123", then lock keystore
...$ keystore_cli password 123                        service call android.security.keystore 8 s16 123
password: No error (1)                                  Result: Parcel(00000000 00000001  '.....')
...$ keystore_cli lock                                service call android.security.keystore 9
lock: No error (1)                                     Result: Parcel(00000000 00000001  '.....')
#
# Attempt to unlock with bad password: System will count down attempts (errors 13,12,11,10)
# then reset (return UNINITIALIZED followed by SYSTEM_ERROR)
...$ keystore_cli unlock bad                          service call android.security.keystore 10 s16 bad
unlock: Wrong password (4 tries left) (13)             Result: Parcel(00000000 0000000d  '.....')
```

输出结果 5-14 使用 keystore\_cli 和 service 两种不同的 keystore 接口

不过，如表 5-13 所示，有许多命令在 keystore\_cli 中并没有实现。不过你仍可以通过调用数字形式表示的功能码，并使用 service 程序的 s16 和 i32 变量标识符（specifier），传递不同类型的参数的方式，使用表中用灰色字体表示的命令。

```
system@generic$ service call android.security.keystore 12 s16 name1 \ # Name
                                                    i32 13 \ # Len
                                                    s16 hello \ # Value
                                                    i32 -1 i32 \ # UID (-1 = caller)

Result: Parcel(00000000 00000001 '.....')
#
# Attempt to retrieve value from keystore
system@generic$ service call android.security.keystore 2 s16 name1
Result: Parcel(
  0x00000000: 00000000 0000000d 00000005 00650068 '.....h.e.'
  0x00000010: 006c006c 0000006f 'l.l.o...')
```

输出结果 5-15 直接用 service call 产生一个 key

keystore 的具体实现（通过硬件抽象模型或可能存在的一个硬件组件）将在第 2 本中讨论。

debuggerd<sup>[64]</sup>

无论开发人员有多努力，他们编写的应用都不可避免地会有 bug，可能会导致应用崩溃。为了修复这些 bug，系统中必须要提供一个有效的机制，来收集应用崩溃时的数据。在桌面系统中，崩溃会产生一个核心转储（core dump）——但是在移动设备中，可能根本就没有这个选项。因为核心转储通常都会很大（动辄几百 MB 甚至更大），而（移动设备中的）存储空间却是非常有限的。退一万步讲，即便能够把核心转储保存下来，把这么大的文件从移动设备中拷出来也不是件轻而易举的事。

Android 中引入了 debuggerd 来解决这个问题，它的作用与 iOS 中的 CrashReporter 类似。这个小小的守护进程在正常情况下一直是处于休眠状态的，直到有某个应用崩溃才会（根据 socket 中的事件）被唤醒。感谢 Android 中的 linker，Android 中的所有进程都会自动安装一个信号处理句柄（signal handler），以处理表 5-15 所列出的各种出错信号。

表 5-15 会被 debuggerd 接收的各种信号

信 号	全 称	产生原因举例
ILL	非法指令	错误的机器码
TRAP	调试陷阱（Trap）	断点
ABRT	主动中断（Voluntary Abort）	断言失败
BUS	总线错误	MMU 出错
FPE	浮点异常	除零错
SEGV	段违规	使用 NULL 指针
PIPE	管道破裂（Broken Pipe）	在管道（pipe）的另一端读数据的进程突然中止



无论是收到表 5-15 中的哪个信号,对它的处理方式都是一样的——调用 `debugged_signal_handler`, 这个函数负责通过对应的 socket 与 `debuggerd` 建立连接,并给它发送一个消息。这个消息会唤醒 `debuggerd` 守护进程,并使它生成一个 **tombstone**。这个所谓的“tombstone”实际上就是一个崩溃报告 (crash report),它是 `debuggerd` (使用 Linux API `ptrace(2)`) 通过附加 (attach) 到出错的进程上,获取崩溃时产生的信号,并查看该进程内存中的数据的方式产生的。以这种方式,而不是产生一个完整的核心转储, **tombstone** (有望) 可以抓到造成崩溃的本质原因,并进行基本的崩溃处理 (crash processing)。 **tombstone** 会被保存到 `/data/tombstones` 目录中,其中也可能包含虚拟机级的数据,以及 `ActivityManager` 的 `NativeCrashReporter` 通过 `/data/system/ndebugsocket` 向 `debuggerd` 输出的数据。

如果系统属性 `debug.db.uid` 的值被设为当前崩溃的进程的 `uid`,那么 `debuggerd` 就会把该进程暂停在它最终崩溃的那一瞬间,并等待用户启动 `gdbserver` (并连上来调试这个进程)。它还会在日志中记录一条消息,你可以很方便地在 `logcat` 中看到这条消息 (见代码清单 5-21)。

代码清单 5-21 debuggerd 提交的 Android log 消息

```
I/DEBUG ( 24): *****
I/DEBUG ( 24): * Process pid has been suspended while crashing. To
I/DEBUG ( 24): * attach gdbserver for a gdb connection on port 5039
I/DEBUG ( 24): * and start gdbclient:
I/DEBUG ( 24): *
I/DEBUG ( 24): *      gdbclient app_process :5039 pid
I/DEBUG ( 24): *
I/DEBUG ( 24): * Wait for gdb to start, then press HOME or VOLUME DOWN key
I/DEBUG ( 24): * to let the process continue crashing.
I/DEBUG ( 24): *****
```

`debuggerd` 使用 (将在第 2 本中讨论的) Linux 底层的 `EV_*` 系列 API,等待用户按下某个实体键后,点亮手机上的呼吸灯 (变成红色),以引起用户的注意,表示开始调试。

在 64 位系统中,还会出现一个 `debuggerd64` 程序,以便处理不同指令集、内存布局 (memory layout) 和 ABI 的程序。在第 2 本中,我们将从更一般的角度讨论调试,并更细致地讨论 **tombstone**。

### 实验：在 Android 中重新启用 core dump

Android 中的 **tombstone** 能保存应用程序崩溃时的一部分内存数据,并提供一份简明的崩溃细节信息。不过 Android 并不会让你再有机会去生成一个 **core dump**。可是在调试程序的崩溃原因时, **core dump** 却比 **tombstone** 有用的多。`debuggerd` 守护进程是在 Bionic 层上,通过获取信号发现是否有进程崩溃的,这一事实使我们要完成抓取 **core dump** 的任务变得有些困难——但也不是变得完全不可能。你所要做的只是:在自己的程序代码中增加一个用于信号处理的函数,用它替代程序的默认处理函数,在这个函数中调用 `signal` (或者,如果你喜欢的话调用 `sigaction`)



系统调用，就可以了（见代码清单 5-22）。注意：由于我们要调用的是一个系统调用，所以这一操作并不能在 Java 层上完成。因此我们只能耍一个小花招：在一个很小的 JNI 库中执行这一操作（比如在这个 JNI 库的 `JNI_OnLoad` 函数中）。

代码清单 5-22 一个产生 SIGSEGV 信号，从而生成 core dump 的自杀程序

```
#include <sys/signal.h>

void main (int argc, char **argv)
{
    // restore original behavior, undoing sionics signal handlers
    signal (SIGSEGV, SIG_DFL);

    char *c = NULL;
    c[1] = 'l'; // Harakiri
}
```

编译并运行这个程序，这个进程就会崩溃，但不会生成一个 tombstone。不过，在能够拿到记录了更丰富信息的 core dump 之前，你必须在 shell 中输入 `ulimit -c unlimited` 命令。这是因为在系统默认的资源限制中，对 core 的资源限制是 0（因而也就有效地禁用了它）。另外，还有一个小问题，即这类程序不能在根文件系统（`initrd`）中运行，因为根文件系统是只读的。

如果你是在一个可写的目录中运行这个程序的（对于使用 adb 会话操作 Android 系统的情形而言，`/data/local/tmp` 是个不错的选择），那么默认的行为是：在目录中生成一个名为“core”的文件。不过一旦 Linux 默认的 core-dump 行为启用后，只需设置一下 `kernel.core.pattern`（需要有 root 权限），就能很方便地对输出文件的文件名进行调整（通过 `sysctl` 或 `/proc` 伪文件系统）。

只需（在 Linux 系统中）查阅一下 `man 5 core` 就能获得对 core 参数的详细解释。如输出结果 5-16 所示，常用的方法是给输出文件指定一条完整的路径——用特定的 % 参数，在路径中填入获取到的进程中的元数据。指定完整的路径也可以确保所有的 core dump 都可以集中存放在一个（你可以确保是可以写的）目录中，同时也可以方便你定期整理。输出结果 5-16 就是手把手地教你如何实现上述操作步骤。

```
#
# Default behavior: no tombstone, but no core
shell@flounder:/ $ /data/local/tmp/coretest
Segmentation fault
shell@flounder:/ $ ulimit -c unlimited
root@flounder:/ # su; cat /proc/sys/kernel/core_pattern
core
#
# Default core is created in working directory - but / is mounted read only! So we cd:
shell@flounder:/ $ cd /data/local/tmp; ./coretest
Segmentation fault (core dumped)
# voila!
shell@flounder:/data/local/tmp/ $ ls -l core
-rw----- 1 shell shell 4636672 2016-07-25 21:14 core
#
# Set core directory to /data/local/tmp, and append process id and executable name
root@flounder:/data/local/tmp# echo /data/local/tmp/core.%p.%e > /proc/sys/kernel/core_pattern
root@flounder:/data/local/tmp# ls -l /data/local/tmp/core.*
-rw----- 1 shell shell 4636672 2016-07-25 21:18 core.11700.coretest
```

输出结果 5-16 生成一个传统的 Linux core dump



在 `core_pattern` 文件的第一个字符前加上一个管道命令符(`|`), 将会让你能在 `core dump` 产生时, 运行任意一个程序, 并将当前生成的这个 `core dump` 作为这个程序的标准输入, 从而让这个程序能够以任意方式处理 `core dump`! `debuggerd` 可以 (而且应该可以) 用这种方式实现。

## gatekeeper (Android M)

`gatekeeper` 是 Android Marshmallow 版中新增的一个守护进程, 其作用是: 通过使用可信执行环境 (TEE, Trusted Execution Environment) 增强 Android 在存储 `passcode` 方面的安全性。可信执行环境 (在 ARM 处理器中是由 `TrustZone` 实现的) 使我们能在独立于主操作系统的一个独立的环境中运行一个微型操作系统。“独立性”使得该环境是高度可控的, 并且能确保运行在其中的代码的完整性。“安全世界”与“真实世界”之间的接口通常是由 `mailsolts` 之类的消息传递机制实现的。因此这个“安全世界”可以被用来以单向的方式 (也就是只进不出的方式) 存储秘密信息, 即只能存储这些秘密信息, 但却不能把它们提取出来。不过, 运行在 TEE 中的代码还是可以完全访问这些秘密信息的, 所以我们仍然能通过 TEE 完成数据加密、身份认证之类的工作。

在 `gatekeeper` 中, 会把 `passcode` (图形锁、PIN 码和口令) 加密为能直接被可信执行环境“接受”的 `blob` (二进制数据块)。生成这个 `blob` 的过程需要一个存储在硬件中的密钥的共同参与, 加密后的结果就被存放在 `/data/misc/gatekeeper` 中。这个被存储下来的 `blob` [实际上仍是口令的 Hash (`passcode`)] 还会进一步用文件的访问权限加以保护, 不过即使没有这一层保护, 也没什么大不了的。因为解密或认证的过程需要有硬件的参与 (因此实际上在 Android 中是不会产生 `passcode` 的明文的)。

这类实现被认为比另一个方案 [把口令的 Hash (也就是 `passcode`) 直接存储在文件系统的某个地方] 更安全。因为一旦这个 Hash 被攻击者读取了出来, 他只需做一次暴力攻击或字典攻击, 就能把口令破解出来。而且这类攻击可以离线 (offline) 进行, 甚至可以用分布式计算的方式获得更高的破解效率。但是一旦解密的过程涉及了硬件, 就不光能够强制要求这类操作必须在本机上完成, 同时也能限制解密请求的发送频率和次数——这就能大大减缓暴力攻击的速度。这一举措让 Android 几乎达到了 (尽管还不是平分秋色) iOS 的高强度的加密标准——在 iOS 中, 一个存储在芯片硬件中的密钥 (苹果称之为 “UID key”) 使得暴力破解必须在本机上才能进行, 而且速度很慢, 即使是破解一个只有 4 位数的 PIN 码也要花很长的时间——这一事实在 2016 年 3 月闹得沸沸扬扬的 FBI 与苹果公司数据解密之争中被证明无疑。

`gatekeeper` 是通过 `Binder` 向外提供服务的。这个守护进程导出了一个名为 `android.service.gatekeeper.IGateKeeperService` 的接口, 而它的主要客户端 (毫无疑问) 就是 `LockSettingsService`。

到目前为止，这个守护进程并不向开发者提供任何 API，而是被整合在了前文已经讨论过了的 keystore 服务中。

谷歌在 Android 源码网站<sup>[5]</sup>上提供了 gatekeeper 的相关文档。gatekeeper 的具体实现是留给各个设备制造商的，因为谷歌自己也只为它提供了一个 HAL（硬件抽象层）模块的支持（见图 5-12）。在谷歌提供的这个 HAL 模块中，是把 TrustZone 作为它自己“可以信赖的”TEE OS 的一部分的。gatekeeper 在硬件抽象层上的接口，我们将会在第 2 本中予以详细讨论，因为它在 HAL 上有多层抽象。

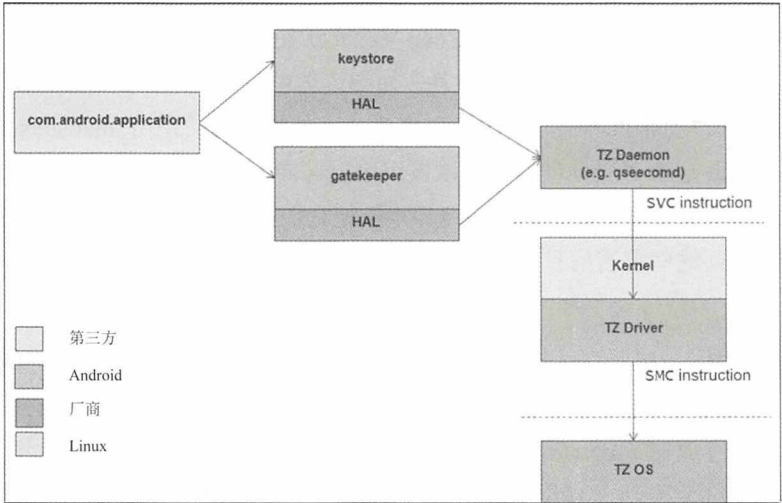


图 5-12 HAL 层上的硬件加密和 TrustZone 接口

## sdcard

并非所有的 Android 设备都必须支持 SD 卡，不过在 Android 系统中还是有一个 sdcard 守护进程，提供用户态中对 SD 卡的支持，其中包括在不支持权限管理的 FAT 文件系统上强制使用权限管理。这是由一种被称为 FUSE（File systems in USEr mode，用户态下的文件系统）的机制来实现的。这一机制在内核中注册了一个 stub 文件系统，并把对该文件系统的所有调用都传递给一个用户态进程（也就是 sdcard 进程）予以处理。相比在内核态中实现一个文件系统，使用 FUSE 更加灵活也更加可靠。文件系统代码使用的灵活性，再加上能够抵御可能并不值得信赖（甚至可能是恶意）的代码对数据结构的破坏，这些优点使得 FUSE 对于那些使用相对不很频繁的文件系统来说是一个很好的选项（不过因为从内核态将数据传到用户态，处理后再将数据传回内核态，这一过程会导致很大的性能开销，所以在其他一些情况下，使用 FUSE 会导致性能极其低下）。



图 5-13 显示了一个文件系统请求从用户态的客户端发往内核,经过 FUSE 被重定向到 sdcard 守护进程,经过处理后,最终原路返回给发起请求的原始客户端的整个流程。

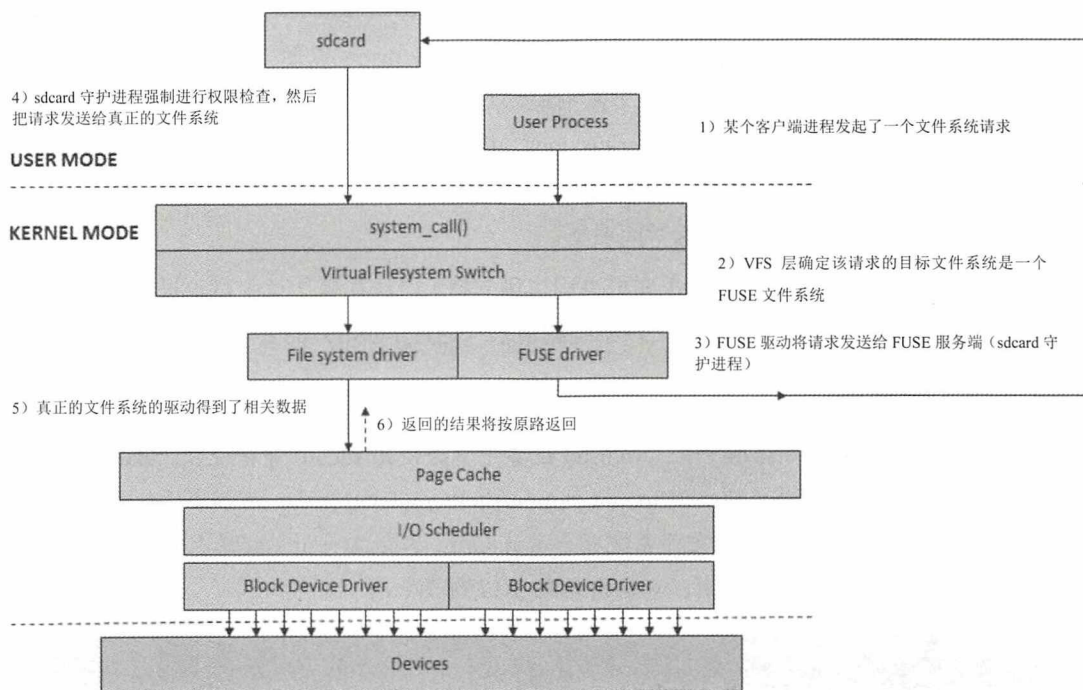


图 5-13 打哪儿来回哪儿去: sdcard 守护进程及其相关操作 (通过 FUSE)

sdcard 守护进程所接受的命令行参数如表 5-16 所示。

表 5-16 sdcard 守护进程所接受的命令行参数

参 数	用 途
-u <i>uid</i>	指定 sdcard 守护进程以哪个用户 ID 的身份运行, 它也是对应文件系统的所有者。通常这个 uid 为 1023 (AID_MEDIA_RW)
-g <i>gid</i>	指定 sdcard 守护进程以哪个用户组 ID 的身份运行, 它也是对应文件系统的所有者。通常这个 uid 为 1023 (AID_MEDIA_RW)
-l <i>path</i>	指定文件系统真正的 mount 点 ( <i>path</i> )
-t #	指定线程数 (默认值是 2)
-l	指定本次 mount 是一个模拟的 (legacy/emulated) mount, 也就是设备并没有真的装有 SD 卡

续表

参 数	用 途
-d	从路径中继承权限
-s	为 media、av 等分割权限

在处理了命令行参数之后，sdcard 会打开设备节点/dev/fuse（以便与内核进行通信），并调用系统调用 mount(2)去 mount FUSE 文件系统，这次系统调用的参数是写死了的，它是：`fd=/dev/fuse fd,rootmode=40000, default_permissions, allow_other, user_id=, group_id=gid`。在完成了降低自身权限的操作之后（使用-u/-g 指定 GID），守护进程会调用 `ignite_fuse()`，进入一个消息循环中，随时准备处理/dev/fuse 发来的请求。

SD 卡中使用的文件系统（由这个守护进程提供支持）已经在第 2 章中讨论过了。下面这个实验将进一步以实例说明，对 SD 卡文件系统的请求（通过 FUSE）的处理流程。

实验：观察 SDCard

无论有没有真的装了一张 SD 卡，Android 设备中总会使用 sdcard 守护进程。/data/media 目录会通过一个 sdcard 守护进程的实例 mount 到/mnt/shell/emulated 上去。而如果设备上还安装了一张实体 SD 卡，这张 SD 卡上的文件系统也会被 mount，尽管这一 mount 请求会导致系统中再多出一个 sdcard 守护进程的实例来，如输出结果 5-17 所示。

```
#
# Use mount to view all mounted file systems, but isolate only FUSE ones
#
shell@htc_m8wl:/ $ mount | grep fuse
/dev/fuse /mnt/shell/emulated fuse rw,nosuid,nodev,relatime,user_id=1023,group_id=1023, ...
/dev/fuse /storage/ext_sd fuse rw,nosuid,nodev,relatime,user_id=1023,group_id=1023, ...
#
# busybox's ps applet will show you the full command line (or you can cat -tv /proc/.../cmdline)
#
shell@htc_m8wl:/ $ busybox ps | grep sdcard
 844 1023 0:02 /system/bin/sdcard -u 1023 -g 1023 -l /data/media /mnt/shell/emulated
1599 1023 0:10 /system/bin/sdcard -u 1023 -g 1023 -w 1023 -d /mnt/media_rw/ext_sd /storage/ext_
```

输出结果 5-17 观察 sdcard mount 的 FUSE 文件系统

实时观察 sdcard 的行为还是有那么一点技巧性可言的。相关的方法在这一章里已经演示了好几遍了。在这里我们仍然可以使用无所不能的 strace 完成这一任务。不过如果你闭着眼睛直接去 trace 主线程的话，可能什么也得不到。sdcard 中无论那个线程都可以为 FUSE 发来的请求提供服务，这就意味着你必须先去/proc/\$SDCARD\_PID/task 目录里看一下创建了哪些线程，然后再用 strace trace 它们（之前我们用得挺称手的-f 参数这里也派不上用场，因为线程是在 strace 附加上来之前就已经创建好了的）。一个不错的经验是同时使用两个 adb 会话，一个用来 trace

sdcard 的线程, 另一个用来对已经 mount 上来了的 FUSE 文件系统进行操作 (比如执行一条 `ls -l` 命令)。这样你就会看到, 在客户端发出请求时, 你发往 `/dev/fuse` 或从 `/dev/fuse` 接收到的数据实际上是一定会经过 sdcard 这个代理的, 必须由 sdcard 把它们再转发给更低层的系统调用。比如在输出结果 5-18 中给出的, 就是在执行 `ls -l /storage/ext_sd` 命令时, 在另一个 adb 会话中 strace 捕获到的结果。

[illegible]

输出结果 5-18 跟踪执行 ls -l 命令时 sdcard 进程的行为

Zygote<sup>[64]</sup>

尽管无论是根据首字母排序还是在本章的编排结构,Zygote 都落在了所有服务排次的最后,但这并不代表它就是不重要的。事实上,Zygote 服务以提供了一台初始化完毕的空 Dalvik 虚拟机的形式(就差加载 main 类了),对所有 Android 框架运行时服务都提供了核心支持。Zygote 服务在/init.rc 中的定义如代码清单 5-23 所示。

### 代码清单 5-23 Zygote 在/init.rc 中的定义

```
service zygotest /system/bin/app_process -Xzygotest /system/bin \
--zygotest --start-system-server
class main
socket zygotest stream 660 root system
onrestart write /sys/android_power/request_state wake
onrestart write /sys/power/state on
onrestart restart media
onrestart restart netd
```

如上面这个代码清单所示, Zygote“真正的名字”是 `app_process`。不过不管怎么说,“Zygote”被接受得更广泛,某种程度上已经成了这个进程的“绰号”了。就像 Zygote 在生物学意义上的含义一样<sup>1</sup>, 这个进程也有着无限多种发展的可能性, 它可以加载任何给定的 Dalvik 类, 并且

1 单词“zygote”在生物学中表示的是“受精卵”的意思。——译者注



可以将进程的拥有者切换为任意用户，不过这类演化是个不可逆的单向过程（这一点又和 Zygote 在生物学意义上指代的东西有几分相似了）。命令行中余下的部分是传进来的参数，在这些参数中，除前面缀有“--”的那两个以外都是直接传给 Dalvik 虚拟机的。而最后那两个参数则是传递给 `app_process` 进行处理的，它们的作用是令虚拟机加载 Zygote 类，并 `fork()` 出一个被用作 `system_server` 的新进程来。

`system_server` 进程（这个进程将在下一章中详细讨论）会继续加载所有的 Android 运行时框架，而 Zygote 将会去绑定（bind）它的 socket（`/dev/socket/zygote`），以监听外部发来的请求。当有新的请求发过来时，这个请求中应该包含一个要求加载的类的类名，于是 Zygote 只需 `fork()` 出一个新进程，并在新进程中加载指定的类就行了。这样就完成了一个新 App 的启动过程，一个新的“生命”就又诞生了。不过，从 Linux 的角度看，所有这些 App 甚至包括 Zygote 本身都只不过是 `app_process` 的实例而已，只不过把它们重命名成了各自对应的名称罢了（你可以执行“`ls -l /proc/pid/exe`”命令验证这一点）。

由于 Zygote 可以“变身”为任意进程，所以它必须保持随时能调整各种选项的能力。有鉴于此，它保留了它自己的 root 权限，拥有权能集（set of capabilities）中的所有权能。不过在 `fork()` 之前，Zygote 会放弃所有的权限，然后调用 `setuid(2)/setgid(2)` 将自己的所有者设为目标 App 的 AID。由于这一操作将在加载 App 的代码（既包括 Dalvik 虚拟机中的代码，又包括 App 中可能存在的原生代码）之前完成，所以可以认为它是安全的。不过 Zygote 在过去的几年中仍然还是曝出了一些漏洞，比如由于没有检查 `setuid(2)` 的返回值而导致的 Froyo 的 Zysploit，以及离现在更近些的时候（2013 年）曝出的 fork-bomb 拒绝服务攻击。

更进一步，我们可以由此推出：所有的 App 都一定是 Zygote 的子进程，唯一的例外是那些直接通过命令行直接调用 `app_process` 的情况（比如，使用第 2 章中提及的 `upcall` 脚本）。你可以自己动手，在你自己的手机上运行一下 `ps` 命令来验证这一点：有些进程（比如本章讨论的所有守护进程）是 `init`（PID 是 1）的子进程，而其他许多进程则是 Zygote 的子进程。这些进程的 PPID 一栏中记录的 PID 是另一个数字，我敢打赌，这个数字一定是 Zygote 的 PID。

## Zygote 背后的逻辑

读到这里你可能不禁要问：不就是启动一个新的 App 嘛，有必要搞得这么复杂吗？还真有必要！因为这样做有不少好处，还不止是在一个方面，这些好处体现在两个方面。

- 应用启动所需的时间会被大大缩短：不论是哪个 App，所有的虚拟机都必须按照某种一样的而且是确定的方式初始化。加载 App 的类只不过是这个操作的最后一步罢了，但是这一操作产生的实际开销却主要是花在加载大量构成了 Android 各种丰富的框架的运行

时类上的。如果你把 App 的加载想象成一场长跑赛，那么使用 Zygote 让 Android 可以在临近终点时才突然插进比赛，只用跑最后一个冲刺即可——这样就省去了前面加载各种库的时间，耗时当然会相对短很多。

- **优化共享内存：**因为所有的虚拟机都是从 Zygote 这里 fork() 出来的，所以它们无疑能够享受到由内核实现的内存共享的优势。特别是，尽管每个 app\_process 的实例都有它自己的虚拟内存，但这些内存中的大部分是只读的（因为其中存放的是类的实现代码），在物理内存中只需保留一份，让各个进程共享就够了。而内存的其他部分（存储的是类的数据，读写）也只要在绝对必要的时候才需要分配一些内存页来存储这些数据 [这种技术叫“写时复制”（copy-on-write）]。因此，大多数 Android 应用都和其他应用（其中还包括 system\_server，它是第一个真正的完整虚拟机的实例）共享了 80%~90% 的内存。这种被最大化了的内存使用方式，使得即使是在那些拥有物理内存相对较少的设备中仍可以“塞”进不少应用。如果现在直接翻到输出结果 7-14 那里，你可以看到一个这样的例子——在这个请你亲自动手完成的实验中，演示了 procrank 和 librank 这两个能够提供内存实时使用情况信息的工具的用法。

这是 Zygote 特有的设计，这一设计使得它在 Java 跌倒的地方获得成功。尽管在虚拟机自身架构方面还有其他的优化（比如，保持引用计数与对象相互分开），但这些优点我们要留待第 2 本中，从程序员的视角再予以深入讨论。同时，在第 2 本中，还会一步一步地向你展示应用启动的整个流程。

所有的虚拟机都是从 Zygote 这里 fork() 出来的，这种优化也不是没有缺点：就像我们将要在第 8 章中讨论的那样：所有的应用都是从同一个进程 fork() 那里出来的，这会有效地破坏地址空间布局随机化（ASLR，Address）——而这一技术却又是对抗代码注入攻击的重要安全手段。换句话说，需求量大的战胜了需求量小的，对性能的渴求压倒了安全需求。最近的学术研究建议把 Morula<sup>[5]</sup> [这还是一个生物学概念，zygote（受精卵）“卵裂”分裂后的胚胎就是“Morula”（桑葚胚）] 作为 Zygote 的一个替代方案——尽管这一方案仍有影响地址空间布局随机化的问题，但它也已经在 Android 中占有一席之地了。

在使用 ART（Android RunTime，Android 运行时）后，Zygote 甚至变得更有效了，因为所有预加载类也都已经经过预编译了。不过麻烦的事也接踵而至，因为 32 位和 64 位地址空间布局并不兼容。

## Zygote 32 和 Zygote 64

在转移到 64 位计算机上的同时，保持对 32 位程序的兼容也是必须的，所以现在 Android

中有两个版本而不是一个版本的 Zygote。在 64 位体系结构中，“zygote\_second”是一个 32 位的进程，它是作为 app\_process32 的实例启动的。由于 64 位的 Zygote 已经把 zygote socket 占用掉了，所以“zygote\_second”还需要另外一个 socket，这一切都如代码清单 5-24 所示。

代码清单 5-24 Zygote 32 在/init.zygote64\_32.rc 中的定义

```
service zygote /system/bin/app_process -Xzygote /system/bin \
service zygote_secondary /system/bin/app_process32 -Xzygote /system/bin \
                                     --zygote --socket-name=zygote_secondary
class main
socket zygote_secondary stream 660 root system
onrestart restart zygote
```

对于自己到底是哪个 zygote 的实例，这一点对于用户 App 来说完全是透明的，尽管不同的 Zygote 会加载不同的库。于是很自然，使用的 JNI 也是不同的，32 位的 Zygote 实例使用的是 /system/lib，而 64 位的 Zygote 使用的则是/system/lib64。查看进程的内存地址空间映射情况（可以使用 cat /proc/pid/maps 命令），也会发现二者在内存使用方面的差异，详见第 7 章的进一步讨论。

本章小结

这一章中讨论了 Android 的各个原生（native）服务，这些服务都是根据/init.rc 中的各个 service 语句块从 init 进程那里 fork 出来的守护进程。这些原生进程负责完成各种杂七杂八的事务性操作，并提供对系统框架的底层支持。

不过框架服务（framework service）就完全是另一回事了——由于这些服务的巨大数量以及需要详细讨论相关细节，我们把相关讨论留在第 2 本中进行。不过在下一章中，我们还是将初步介绍一下它们，通过概览服务的架构，详细说明 servicemanager 和 system\_server 的方式。system\_server 也是所有框架服务的容器，它还会从 bootanimation 那里接管 UI。

本章讨论涉及的文件

小节标题	文件/目录	内 容
adb	system/core/adb	adb 的实现代码，即包括客户端的，也包括服务端的
	f/b/s/ja/com/and/ser/usb/UseDebuggingManager.java	USB Debugging Manager 服务，这个服务实现在 system_server 中
vold	f/b/s/ja/com/and/ser/MountService.java	Mount Service Manager 服务，这个服务实现在 system_server 中



续表

小节标题	文件/目录	内 容
debuggerd	/system/core/debuggerd	debuggerd 的源码
installd	f/native/cmds/installd	installd 的源码
bootanimation	f/base/cmds/bootanimation/BootAnimation.cpp	bootanimation 的源码
sdcard	sys/core/sdcard/sdcard.c	sdcard 的源码

## 参考文献

- [1] Android Full disk encryption:  
[http://source.android.com/devices/tech/encryption/android\\_crypto\\_implementation.html](http://source.android.com/devices/tech/encryption/android_crypto_implementation.html)
- [2] Android Explorations: <http://nelenkov.blogspot.com/2012/08/changing-androids-diskencryption.html>
- [3] BootAnimation collection: <http://www.NewAndroidBook.com/bootanimations/>
- [4] Android Developer, Keystore Documentation:  
<http://developer.android.com/reference/java/security/KeyStore.html>
- [5] Georgia Tech, "From Zygote to Morula":  
<https://taesoo.gtisc.gatech.edu/pubs/2014/morula/morula.pdf>
- [6] Android Internals::A Confectioner's CookBook (Volume I)  
Android Source, "GateKeeper":  
<https://source.android.com/security/authentication/gatekeeper.html>

## 第6章

# 框架服务的架构

上一章中只讨论了一部分 Android 运行时服务，这些服务的共同特点是：它们都是原生级（native-level）的服务，也就是说，这些服务都是用 C/C++ 来实现的，而且在 Java 层上也没有直接可用的编程接口。所以，它们应该归类为支持操作系统自身的服务。不过 App 使用的却是完全另一个服务集（set of services）中的服务，这些服务是由 Dalvik 虚拟机级的框架提供的，带有特定接口。这些服务都有一个 Java 语言的接口，而且其中的大多数是运行在同一个进程（system\_server）的上下文环境中的，并且可以在 servicemanager 的帮助下被找到。

servicemanager 和 system\_server 在上一章中都已经介绍过了。servicemanager 是在讨论 core 类的服务的章节中介绍的，而 system\_server 则是 Zygote 的一部分——因为它是以 --start-system-server 参数从 Zygote 中 fork() 出来的。不过我们确实也应该更深入地讨论这两个东西，因为正是这两个东西联手提供了对整个 Android 框架服务架构的支持和上下文环境，而这就是本章要讨论的。

我们首先要重新深入探究 service manager，这个服务扮演着一个端点映射器（endpoint mapper）的角色（也就是让服务能被找到并被请求）。通过在 servicemanager 中注册，服务就能让客户端看到自己，而客户端也能根据这些注册信息，通过 servicemanager 请求获得一个指向指定服务的连接（或句柄）。由于所有的框架服务都是以同一种方式被调用的，所以接下来讨论的就是这一服务调用模式。我们特别会介绍两个关键组件——AIDL 和命令行工具 service。AIDL 是“Android 接口定义语言”（Android Interface Definition Language）的缩写，服务就是通过它来导出接口（以及服务的各个 API）的。而 service 这个工具则让你能够在命令行下测试和调试这些接口。

在 Android 中，与服务通信（实际上也就是 App 之间的通信）时，使用的底层传输机制是 Binder，应用可以通过 /dev/binder 来访问它。尽管 /dev/binder 看上去只是一个简单的设备节点而

已，但事实上它的背后是个精心设计的 IPC（进程间通信）框架，它不光承担了分发消息的任务，而且还能传递各种对象、描述符等数据结构，此外它还提供了可靠性和安全性保证。这些我们都将在近距离窥视这个服务的内部实现时予以讨论。

最后，我们还要看一下 `system_server` 本身。这个进程的作用是为其他服务提供一个宿主进程，其他服务大多是以这个进程中的线程的形式实现的<sup>1</sup>。我们将详述这个重要的进程的启动，操作以及内部结构。而对于各种具体的服务，我把它们留待本书的第2本中再详细讨论。

## 6.1 再探 `servicemanager`

回顾一下上一章，你就会发现，这个叫 `servicemanager` 的服务被 `init` 归类在“core”类中。其他一些关键服务都依赖于这个服务，如果它崩溃了，这些服务都必须重启。更夸张的是，`servicemanager` 是由 `critical` 关键字修饰的，也就是说，如果这个服务重启后又崩溃了，`init` 会不断地重启它。要是这样还是不能解决这个问题，系统就会重启或者进入 `recovery` 模式。

使 `servicemanager` 如此极端地重要的原因在于它的功能：它是作为其他操作系统服务的定位器（locator）或称索引目录（directory）而存在的。任何一个应用或系统组件想要使用其他服务（完成某个操作）时，都必须先到 `servicemanager` 这儿来查询，获得一个句柄（handle），然后才能继续执行操作。类似地，服务也没法直接告诉客户端自己在哪儿，它也必须先到 `servicemanager` 这儿注册一下，然后才能被需要使用它的客户端找到。据此我们也不难得出结论：如果这个 `servicemanager` 重启了的话，所有依赖于它的服务都必须重启。毕竟，`servicemanager` 重启也就意味着服务的索引目录也必须从零开始，重新构建了。所以所有的服务都必须重新再注册一遍。甚至还可以想见更进一步的结论：如果 `servicemanager` 挂了的话，进程间通信（IPC，Inter-Process Communication）也将无以为继。

Android 中使用的进程间通信模型将在本章中的稍后部分予以讨论，不过现在，我们只要知道它是由一个专门的内核组件“binder”提供的就行了。用户模式的服务是通过一个字符设备节点——`/dev/binder`（这个设备节点可以被所有的进程读取和访问），来访问 `binder`，从而进行进程间通信的。不过同一时刻，只有一个用户态进程可以使用 `Binder` 中的 `context manager`，`context manager` 是系统中所有其他进程（无论它是客户端进程还是服务端进程）的汇接点。服务必须把自己的服务名以及接口注册在 `context manager` 中，而客户端也必须通过在 `context manager` 中进行检索，找到它们要使用的服务。

---

1 `SurfaceFlinger` 和 `media` 服务是值得注意的少数几个例外，另外请注意，应用中提供的（第三方）服务是运行在应用自己这个进程中的。——原注



因此 `servicemanager` 也只是一个很小的二进制可执行文件，所完成的操作也很简单：它首先调用 `binder_open` 获取 `/dev/binder` 的文件描述符，然后再调用 `binder_become_context_manager` 注册 `ServiceManager` 服务，该服务的 `handle` 索引值为 0。再接下来，`servicemanager` 就会进入一个名为 `binder_loop` 的无限循环中。在这个无限循环里，`servicemanager` 会让自己进入阻塞状态，直到 `/dev/binder` 中产生一个 `transaction`（也就是某个客户端发来了请求），随后进程就会被唤醒，并调用它的 `svcmgr_handler` 回调函数处理这个 `transaction`。

服务的查找在某种程度上有点类似于自助式服务——换言之，`servicemanager` 必须是全局可访问的。只有这样，各种服务才能上它这儿来注册，客户端也才能来查询服务。在使用 C/C++ 代码时，服务和客户端同样可以调用 `defaultServiceManager()` 获得一个访问 `service manager` 的句柄（从技术上说，这只是它的接口，是个 `sp<IServiceManager>` 类型的对象），这个定义在 `IServiceManager.h` 中的接口，导出了几个 `transaction` 请求码。表 6-1 中列出了这些请求码，及实现这些请求码的可供 C/C++ 代码调用的 API。值得注意的是，在这张表中并没有用来删除服务的 API，这是因为在对应的进程死掉之后，服务会被自动删除。因为 `Binder` 能够检测到这种情况，并在检查到这种情况时，向 `servicemanager` 发送一个 `BR_DEAD_BINDER`（进程死亡通知）。

表 6-1 调用 `servicemanager` 各项功能的请求码及对应的 API 方法

请求码	API	注释
<code>SVC_MGR_ADD_SERVICE</code>	<code>addService(name, service, allowIsolated)</code>	供各类服务把自己注册到 <code>service manager</code> 里。服务可以（通过 <code>allowIsolated</code> 参数）决定是否允许被隔离的进程（也就是运行在沙箱中的进程）连上自己
<code>SVC_MGR_GET_SERVICE</code> <code>SVC_MGR_CHECK_SERVICE</code>	<code>checkService(name)</code>	获得指定名称的服务的句柄
<code>SVC_MGR_LIST_SERVICE</code>	<code>listServices()</code>	返回一个记有所有服务的向量（列表），这个 API 并不是给普通框架用的，而是供 <code>service list</code> 命令使用的

`addService` 被认为是一个非常敏感的功能，只有 UID 为 0 或 1000（`AID_SYSTEM`）的服务才能自由地注册服务，其他的系统服务则会受到限制。到了 KitKat 及以后的版本中，这一限制注册的任务又改由源码中一个事先写好的 `allowed` 数组来完成，该数组中的内容如表 6-2 所示。

表 6-2 事先写好的注册服务的限制规则

<code>AID_MEDIA</code>	<code>media.audio_flinger</code> , <code>media.log</code> , <code>media.player</code> , <code>media.camera</code> , <code>media_audio_policy</code>
<code>AID_DRM</code>	<code>drm.drmManager</code>
<code>AID_NFC</code>	<code>nfc</code>

续表

AID_BLUETOOTH	bluetooth
AID_RADIO	radio.phone, radio.sms, radio.phonesubinfo, radio.simphonebook
AID_RADIO <sup>1</sup>	phone, sms, iphonesubinfo, simphonebook
AID_MEDIA	common_time.clock, common_time.config
AID_KEYSTORE	android.security.keystore

到了 Lollipop 版后，这个写死在源码中的数组又被移到了 SELinux 的/service\_contexts 文件中（见代码清单 6-1），这使得系统能以极为灵活的方式对服务进行控制——system\_server.c 这个源码文件已经被简化成对 check\_mac\_perms() 函数的一次调用，这个函数然后又去调用 selinux\_check\_access() 函数。用这一方式，可以对所有服务的注册和查找强制进行安全检查，同时，这也使得设备的制造商可以在不需要重新编译任何源码的前提下添加它们自己的服务。

代码清单 6-1 SELinux 的/service\_contexts 策略文件的内容

```
#line 1 "external/sepolicy/service_contexts"
accessibility                u:object_r:system_server_service:s0
..
android.security.keystore    u:object_r:keystore_service:s0
..
batteryproperties            u:object_r:healthd_service:s0
batterypropreg               u:object_r:healthd_service:s0
..
bluetooth                    u:object_r:bluetooth_service:s0
..
common_time.clock            u:object_r:mediaserver_service:s0
common_time.config           u:object_r:mediaserver_service:s0
..
display.qservice             u:object_r:surfaceflinger_service:s0
..
drm.drmManager               u:object_r:drmserver_service:s0
..
inputflinger                 u:object_r:inputflinger_service:s0
..
media.audio_flinger          u:object_r:mediaserver_service:s0
media.audio_policy            u:object_r:mediaserver_service:s0
media.camera                 u:object_r:mediaserver_service:s0
media.log                    u:object_r:mediaserver_service:s0
media.player                 u:object_r:mediaserver_service:s0
media.sound_trigger_hw       u:object_r:mediaserver_service:s0
nfc                           u:object_r:nfc_service:s0
..
*                             u:object_r:default_android_service:s0
```

编程时使用的 API 已经打包在框架类 android.os.ServiceManagerNative 中了，这个类又由 android.os.ServiceManager 做了进一步的封装。Android 也不鼓励 App 直接使用这个类，而是希望开发者调用 Context.getSystemService() 方法来查找系统服务，并通过 intent 使用第三方服务。不过无论是使用哪种方式，与服务进行通信（不论它是个系统服务还是第三方服务）都是通过

1 这些都曾是合法的服务名，只是在改用了有 radio.\* 前缀的服务名之后，这类没有 radio.\* 前缀的服务名就不再使用了。——原注

binder 消息来完成的。Servicemanager 所扮演的角色只是一个服务目录而已。如图 6-1 所示。

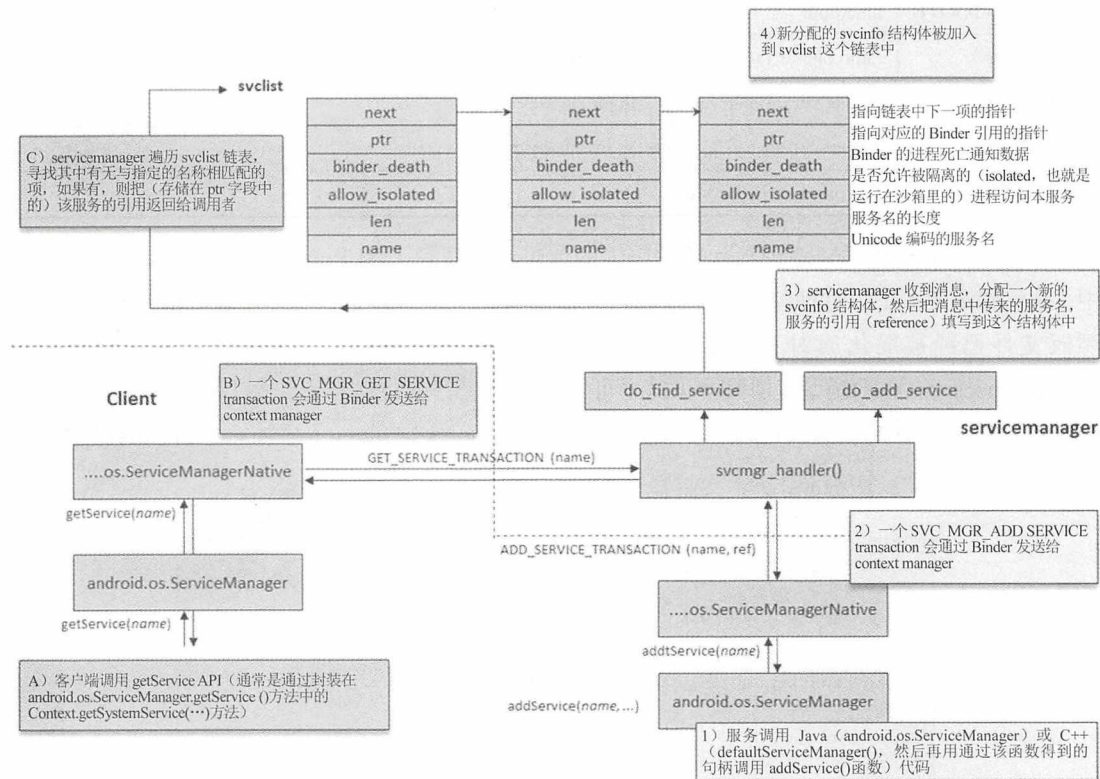


图 6-1 Android 框架服务的注册和访问

## 实验：把 service 命令用作 service manager 的接口

Android 中提供的 `service` 这个命令行工具, 可以用作 `service manager` 的一个简化了的接口。这个简单的程序也可以展示如何使用可以在编程时使用的 API 查询服务, 使用 `service list` 命令, 你可以列出所有已经注册了的服务以及它们公开的接口 (至于怎么使用这些接口, 我们将在本章稍后部分予以讨论), 而使用 `service check` 命令, 我们可以检查给定的服务是不是能够被我们连上。

输出结果 6-1 中显示的就是在一台安装 Android L 版操作系统的 Nexus 5 手机上运行 `service list` 命令的输出结果。不过考虑到在任何一台 Android 设备上你都能很方便地运行这条命令, 但是你得到的结果可能会和我得到的有部分不同, 所以这里我把 L 版中新增的服务以及在模拟器中不会出现的 service 高亮显示了出来。



```

root@generic# service list
Found 93 services: # Emulator shows only 87 services, 75 in KK
1  sip: [android.net.sip.ISipService] # Not in emulator
2  phone: [com.android.internal.telephony.ITelephony
3  iphonesubinfo: [com.android.internal.telephony.IPhoneSubInfo
4  simphonebook: [com.android.internal.telephony.IIccPhoneBook
5  isms: [com.android.internal.telephony.ISms
6  nfc: [android.nfc.INfcAdapter] # Not in emulator
7  telecomm: [com.android.internal.telecomm.ITelecommService] # L
8  launcherapps: [android.content.pm.ILauncherApps
9  trust: [android.app.trust.ITrustManager] #
10 media_router: [android.media.IMediaRouterService
11 tv_input: [android.media.tv.ITvInputManager] #
12 hdmi_control: [android.hardware.hdmi.IHdmiControlService] #
13 media_session: [android.media.session.ISessionManager] #
14 print: [android.print.IPrintManager
15 assetatlas: [android.view.IAssetAtlas
16 dreams: [android.service.dreams.IDreamManager
..
20 voiceinteraction: [com.android.internal.app.IVoiceInteractionManagerService]
21 appwidget: [com.android.internal.appwidget.IAppWidgetService]
22 backup: [android.app.backup.IBackupManager]
23 jobscheduler: [android.app.job.IJobScheduler] # L
...
..
38 ethernet: [android.net.IEthernetManager] # Not in emulator
39 wifiscanner: [android.net.wifi.IWifiScanner] # L
40 wifipasspoint: [android.net.wifi.passpoint.IWifiPasspointManager] # L
41 wifi: [android.net.wifi.IWifiManager]
42 wifip2p: [android.net.wifi.p2p.IWifiP2pManager]
43 netpolicy: [android.net.INetworkPolicyManager]
44 netstats: [android.net.INetworkStatsService]
45 network_score: [android.net.INetworkScoreService] # L
...
55 bluetooth_manager: [android.bluetooth.IBluetoothManager] # Not in emulator
..
87 display.qservice: [android.display.IQService] # owned by SF, Not in Emulator
#
# Use "service check" with one of above names to see if service is alive
#
root@generic# service check media.camera
Service media.camera: found

```

输出结果 6-1 在一台安装 Android L 的 Nexus 5 手机上运行 service list 命令的结果

不难想象，在不同的设备上运行这条命令的结果会有一定的差异。有些差异是显而易见的（比如在平板电脑上就不会有 Phone 这个服务），而另一些差异则不那么容易想到（那些厂商提供的服务或者与系统版本有关的服务）。

IBinder 接口定义了一个 dump() 方法，它是供 dumpsys 命令对服务进行诊断的。如果不输入任何参数，dumpsys 将用和 service list 一样的方法枚举出所有的服务，并把它们依次列出。而在另一些情况下，如果给出了具体的参数，各个服务的检查结果就千变万化了。有些服务还会导出一个 checkin 方法，它可以在 dumpsys -c 或者 --checkin 时被调用。

## 6.2 服务调用的模式

Android 的框架服务都是实现在 `system_server` 的各个线程中的。因此应用调用它们时，必须使用进程间通信（IPC，Inter Process Communication）的方式。这就是 **Binder**（Android 特有的 IPC 机制）发挥作用的地方。应用需要先在自已这个进程中调用 **Binder**，获取一个端点描述符，然后才能与远程服务建立连接。服务中提供的各种方法是通过 IPC 消息进行调用的，这一模式，也被称为远程过程调用（RPC，Remote Procedure Call）。

### IPC? RPC?

术语 IPC 和 RPC 经常会被混用——尽管通常是不正确的。由于这两个概念是接下来讨论 Android 服务的基础，所以有必要在这里把二者的区别讲清楚。

- **进程间通信（IPC，Inter Process Communication）**：这个概念泛指进程之间任何形式的通信行为，是个可以拿来到处套的术语。它不仅仅包括各种形式的消息传递，还可以指共享资源（最明显的就是共享内存），以及同步对象 [mutex 或者其他类似的东西，即确保安全地并发访问共享资源（也就是防止两个或两个以上的对象同时对同一个数据成员进行修改，从而导致的数据被破坏，或者竞争条件下同时读/写数据而导致错误的情况发生）的东西]。
- **远程过程调用（RPC，Remote Procedure Call）**：这个术语特指一种隐藏了过程（方法）调用时实际通信细节的 IPC 方法。（在使用 RPC 时，）客户端将调用一个本地方法，而这个本地方法则是负责透明地与远程服务端（这个远程服务端甚至可以在不同的时间段里是不同的机器）进行过程间通信。这个本地方法会将相关参数顺序打包到一个消息中<sup>1</sup>，然后把这个消息发送给服务端提供的方法，服务端的方法会从消息中解出序列化（deserialize）发来的参数，然后执行，最后仍以这一方式（当然这时发送方和接收方换了个位置）将方法的返回值（如果有的话）发送给客户端。

所以，任何 RPC 机制都一定是 IPC 机制（因为前者只是后者的一种特殊形式），但反过来却不一定是这样。正如我们在这一节中将要讨论和深究的那样，Android 中服务的调用模式是用 RPC 方式实现的。表 6-3 中对比了现代操作系统中使用的 RPC 机制。

---

<sup>1</sup> 这一动作即“序列化”（serialize）。——译者注

表 6-3 各种常见操作系统中 RPC 机制的对比

OS	实现模块	作用范围	索引目录	预处理器	通信信道
UNIX	SunRPC	本地/远程	portmapper	rpcgen	UDP/TCP
OSX/iOS	Mach	本地（远程）	launchd(mach_init)	mig	Mach 消息
Android	Binder	本地 <sup>1</sup>	servicemanager	aidl	/dev/binder

如表 6-3 所示，所有的 RPC 机制都有一些共同特性：

- 作用范围——表示 RPC 是否可以在远程和本地主机间进行，还是只能在本地主机中进行。
- 索引目录——提供定位服务这一查询功能的服务程序。
- 预处理模块——用来产生将参数序列化装入消息或从消息中解出序列化参数的代码的工具。
- 通信信道——消息传递的媒介。

我们将在讲解 Binder 时再来回顾 RPC，并更深入地讨论它。

Android 应用的开发者可以幸福地忽略掉服务调用的底层实现方式。大多数 Android 应用的开发者所熟悉的调用服务的方法是：他们只需调用 Context 对象的 getSystemService()方法，这个方法只需接收某个 Android 系统服务的服务名作为输入参数，就能返回一个具体格式/含义不详的对象，通过这个对象就能得到指定的服务对象，并通过它调用服务的方法。

图 6-2 中展示的就是这个调用大多数服务方法的通用模式。该图已经做了某种程度的简化（例如，系统服务的句柄应该是会被缓存的），但仍足以展示整个过程。服务在被使用之前首先应该已经由某个 server 进程（通常这个进程就是 system\_server，但也有可能是个第三方进程）通过调用 android.os.ServiceManager 中的方法注册好了。我们回想一下，这个类提供了 ServiceManager 的一个 Java 接口。

---

<sup>1</sup> 根据其设计本意，Android 的 Binder 将其作用范围限制在了本地，但是我们很方便就能安装一个本地代理（proxy）进程，进一步对通过 TCP 或 UDP socket 传输的数据进行序列化或解序列化。这样就扩展了 Binder 的作用范围，对于远程访问工具（RAT，Remote Access Tool）来说，这是一个极其有用的功能。——原注



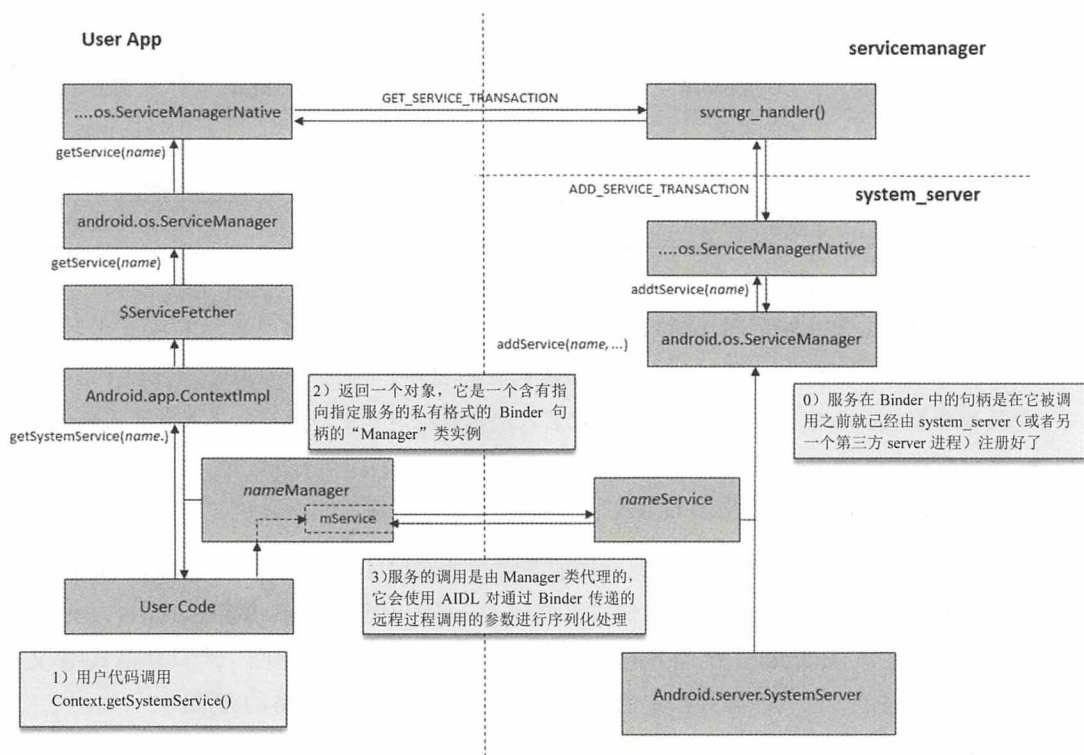


图 6-2 Android 系统服务的调用模式

## 优点和缺点

Android 的系统服务架构遵循的是一个典型的本地客户端/服务端通信模式，它和其他操作系统（比如 iOS）中使用的是一样的。虽然在 iOS 中没有 Binder——它使用的是它自己实现的一个被称为“Mach 消息”（Mach Message）的消息传递架构。在 iOS 中，servicemanager 的角色（即端点映射器）是由 launchd 进程扮演的，此外，这个进程同时还要扮演 Android 中由/init 完成的，传统 linux 中 PID 为 1 的进程所饰的角色。

我们一眼就能看出这一架构的一个明显的缺点：过程间通信的开销实在是太大了，特别是在必须进行的序列化和解序列化消息的过程中，以及在交替切换进程时所必需进行的进程上下文切换时，开销尤为巨大。这一缺点确实会带来可观的性能损失。

既然有这么大的缺点，那么这一架构一定能带来更大的，或者至少是足以抵消这一缺点的好处，才会被采用，事实也确实如此：除了能让设计更为简洁之外，它还能做到权限隔离，这自然也能让客户端/服务端架构获得更高的安全性。在设计所依据的场景中，客户端进程应该是

个不可信的用户 App，它应该完全避免拥有任何权限，所以它想要完成任何操作，都必须完全依赖于服务调用。如果用户 App 是用原生代码写的，这也就等于 App 是运行在沙箱（sandbox）中的，即使在需要时，它也是不能（直接）访问设备或数据存储的（只能通过系统服务来完成这一操作）。事实上，这就是在 iOS 中发生的情况[在 iOS 中 App 是被“关在监狱里的”（jailed），这也是术语“越狱”的由来]，而在 Android 中，情况也差不多，对于大多数进程来说，Android 是根据文件系统中的访问控制权限来决定相关访问是否应该被拒绝的。

而另一方面，服务进程则是可信的，我们指望用它们进行所有的安全检查，在同意对相关请求提供服务之前，确认对应的客户端确实拥有权限。再说一遍，在这一点上 Android 和 iOS 这两种相互竞争的系统的做法是一致的。只不过 iOS 应用用的是（嵌入在二进制可执行文件的代码签名中的）entitlement，而 Android App 用的是 apk 安装包中的 manifest 文件。在这两种情况下，权限的声明都是位于应用运行时的作用域之外的，即它们会在应用安装时被检验（或者在 iOS 中，由苹果公司负责验证），而应用自身是无法修改它们的，特别是 iOS 中使用的 entitlement，它是（作为缓存的代码签名块的一部分）被存放在内核空间中的，而 Android 中各个 App 所拥有的权限则是由 PackageManager 予以维护的。

## 序列化和 Android 接口定义语言（AIDL）

在调用模式的设计的术语中，getService()方法返回的对象只是一个“代理”（Proxy）。在这个对象内部记录着一个通过调用 binder 获得的指向实际服务的引用（reference），而该对象导出的各个方法，在大多数情况下实际上也只是一些 stub 容器而已，这些容器也被称为“Parcel”，其中存放的是被顺序打包（序列化）到 Binder 消息里去的，需要传递给远程方法的各个参数。远程调用的各种方法及其参数就是以这一方式使用 AIDL 序列化的。实际上 AIDL 本身并不是一种真正的语言，它实质上只是一种能被 aidl SDK 程序（在 build 过程中，如遇到.aidl 文件时就会调用它）读懂的 Java 衍生语言而已。aidl 能够自动生成把相关参数序列化打包到 Binder 消息中去，并从返回的 Binder 消息中提取出远程方法的返回值所需的 Java 源码。这些代码被称为“样板文件”（boilerplate），即它可以根据定义文件自动生成，并保证编译得干净利落。.aidl 文件的样例如代码清单 6-2 所示。

代码清单 6-2 .aidl 文件的一个例子

```
package com.NewAndroidBook.example; // Creates java directory structure
import com.NewAndroidBook.whatever; // Dependencies, if any

interface ISample {

    // Published interface - will be shown as com.NewAndroidBook.example.ISample
    // The numbers are the ones used when serializing (and using service call)

    /* 1 */ void    someFunc    (int    someArg); // no return value, integer argument
    /* 2 */ boolean anotherFunc(String someArg); // returns boolean, string argument

    // ... etc.. etc..
}
```

就像你看见的，.aidl 文件有点类似于头文件，其中只定义了方法（即可能存在的对象），但并没有给出它们的具体实现代码。在本书第 2 本讨论各种框架服务时，你会在 AOSP 源码中看到许多更复杂的.aidl 文件。

aidl 工具完成了一项几乎是不可思议的任务：向开发者隐藏了 Android IPC 机制实现的细节。事实上这个任务完成得是如此漂亮，以至于大多数开发者竟然可以心安理得地忽略掉 Binder 所扮演的角色，甚至可以完全无视它的存在。不过了解 Binder 的作用，能给我们接下来对 Binder 的讨论，以及在第 2 本中深入地讨论其内部细节开个好头。

高级用户依然可以把 Binder 丢在脑后，特别是在拥有了像 service 这样强大的工具后。使用 service 我们就能够在命令行中直接调用 Android 服务提供的方法，如下面这个实验所示。

### 实验：使用 service 命令调用服务

上一个实验中，我们只是把 service 用作 servicemanager 进程的一个命令行接口，演示了它的基本用法。不过 service 真正强大的部分在于：它能直接调用各个 service 中的方法。

使用 service call 调用一个方法其实也很方便：只需指定服务名及要调用的方法的序号（这个序号其实就是按各个方法在服务的.aidl 文件中的出现顺序，分配到的一个流水顺序号）即可。此外，根据被调用方法的具体定义，可能还需要输入一些被调用的方法的参数。service 这个命令行程序支持两种类型的参数——i32（用来传递一个 int 型的变量）和 s16（用来传递一个 unicode 编码的字符串）。不过在实际使用过程中，int 型变量可以用来传递任何一种 32 位的变量（比如 float 型变量），而 unicode 编码的字符串也可以用来传递任意一种对象。

所有用 service list 命令列出的服务（见输出结果 6-1）都有一个接口（在服务名后面的方括号中给出），我们可以用上述方法调用它们。在 AOSP 源码中，每个服务都有一个对应的.aidl 文件，服务的所有方法及其参数都明确地定义在这个文件中。有了这些定义，你就可以根据自



己的需要调用其中任何一个方法——数出该方法的序号，并传给它对应的参数即可。表 6-4 给出的是一些你可能感兴趣的 service 的方法。

表 6-4 service call 命令

service call ...	接 口	方 法	产生的行为
phone 2 s16 “foo” s16 “555-1234”	ITelephony	call(String callingPackage, String number)	给指定的号码打个电话
statusbar 1	IStatusBarService	expandNotificationsPanel()	弹出通知框
statusbar 9		expandSettingsPanel()	弹出设置对话框
statusbar 2		collapsePanels()	隐藏所有的面板
dream 1	IDreamManager	dream()	进入屏保（如果已经配置了屏保的话）
power 10(< 4.4.1)	IPowerManager	isScreenOn()	如果屏幕亮着返回 1，否则返回 0
power 11(> 4.4.2)			

 在不同版本的 Android 之间（甚至是在同一个 API 版本号的不同 Android 之间），分配给各个方法的序号可能会发生变化，例如 KitKat 中的 IDisplayManager 和 IPowerManager。尽管这种情况极少发生，但它还是会发生的，所以请小心为妙。一般而言，在程序中使用这种直接写死的序号的方式调用方法并不是个好主意。如果你确实需要在工具或 App 中使用这些私有 API 的话，请用对应版本源码中的 .aidl 文件，在编译过程中自动生成被调用方法的序号。

用这种方法调用一个方法，方法将会把返回结果放在一个 Parcel（Binder 中用来称呼消息的术语）中返回来。每个 Parcel 中至少含有一个 32 位的返回值[用 0x00000000 表示方法执行成功，其他的一些值表示不同的出错码。如果调用时输入的方法序号超出了规定的范围，通常会返回 0xffffffff 或 0xfffff6b（“not a data message”）[根据 AIDL 中该方法返回值的具体定义，在这个 32 位数后面还会跟一个 int 型的数（i32）或一个二进制数据块[格式是一个表示数据块长度的数，再加上二进制数据块本身（通常这个二进制数据块中存放的是一个 unicode 字符串，不过有时也会是其他类型的对象）]。因为 service 和 Binder 一样，对这样一个二进制数据块中放的是什么数据是一无所知的，所以它就没法向 od 命令那样直接把返回结果显示出来，而是只能把 Parcel 中的内容以十六进制的形式显示出来（边上再辅以这些数据的 ASCII 含义）。

只有那些拥有（在方括号中给出的）公开接口的服务可以被调用。注意，也不是所有的服务都会盲目地让自己能够以这种方式被调用：根据安全策略，不同服务的安全策略都是不一样的，你所请求的服务可能会被拒绝。如果发生了这种情况，service call 命令的输出结果中会包含一个 unicode 编码的出错消息，如输出结果 6-2 所示。

```
# Attempt to call cancelMissedCallsNotification(), which requires MODIFY_PHONE_STATE
# (You can get past this, as well as most other permission checks, by running as root)
#
shell@htc_m8wl:/ $ service call phone 13
Result: Parcel(
  0x00000000: ffffffff 00000050 0065004e 00740069 '....P...N.e.i.t.'
  0x00000010: 00650068 00200072 00730075 00720065 'h.e.r. .u.s.e.r.'
  0x00000020: 00320020 00300030 00200030 006f006e ' .2.0.0.0. .n.o.'
  0x00000030: 00200072 00750063 00720072 006e0065 'r. .c.u.r.r.e.n.'
  0x00000040: 00200074 00720070 0063006f 00730065 't. .p.r.o.c.e.s.'
  0x00000050: 00200073 00610068 00200073 006e0061 's. .h.a.s. .a.n.'
  0x00000060: 00720064 0069006f 002e0064 00650070 'd.r.o.i.d...p.e.'
  0x00000070: 006d0072 00730069 00690073 006e006f 'r.m.i.s.s.i.o.n.'
  0x00000080: 004d002e 0044004f 00460049 005f0059 '..M.O.D.I.F.Y._.'
  0x00000090: 00480050 004e004f 005f0045 00540053 'P.H.O.N.E._.S.T.'
  0x000000a0: 00540041 002e0045 00000000 'A.T.E.....'
```

输出结果 6-2 service call 返回的出错消息

不过,一旦绕过了这些权限验证(比如你可以以 root 身份运行程序),以这种方式使用 service call 就能让你拥有近乎无限的能力——能够使用 Android 框架服务提供的所有特性和功能。随着接下来一个个地介绍各种 Android 框架服务,我们将向你展示这些服务在对应的 .aidl 文件中的定义,以及服务中各个方法的序号。

## 6.3 Binder

前文中已经提及 Binder 好几次了,但我们一直仅仅是从高层(而非底层)对它进行讨论的。事实上,从高层角度讲,我们只要知道 Binder 就是一个特殊的文件描述符就够了(尽管它实际上是个连向服务的专用内核驱动)。这实际上也是 Linux 层的视角下的 binder——进程通过 /proc/pid/fd 目录看到的 binder。事实上,系统中的几乎所有进程(除了少数几个原生进程外)都会去打开一个指向 /dev/binder 的句柄。

不过 Binder 的大部分内部工作机制仍然笼罩在一团迷雾里,这可能是因为对于大多数开发者来说,忽略这些底层细节才是明智之选,而对于那些想要了解底层细节的人来说,反正 Android 是开源的……不过就本书的写作目的而言,最好还是掀开迷雾的一角,从更近的距离观察 Binder,使你在无须阅读(没有什么良好文档支持的)源码的前提下了解其功能。

### 简明历史

追根溯源,Android 的 Binder 实际上是源自另一种操作系统“BeOS”中的 Binder 的。在 BeOS 中, Binder 是用于在底层支持 BeOS 中各种丰富多样的框架相互进行内部连接的。不过在被宣布为“下一代操作系统”之后, BeOS 就不再受到关注,粉丝也流失殆尽,最终被 Palm 收购掉了。啊,要是你觉得 Palm 这个名字好像不怎么有名……好吧,这不怪你, Palm 的 Pilots

操作系统是上个世纪末一度十分流行的一个操作系统。在分崩离析陨落尘土之前，它也曾把 3COM 的股价抬到过非常高的位置。Palm 最终被 HP（惠普）并购，其操作系统又被用作了“WebOS”系统的基石，这就又是一段新的“征程”了（不过比起它之前的承诺，还是有一段不小的差距）。

不过，Binder 幸存了下来。除了继续作为 PalmOS 的一部分（而且还被并入了 Palm 自己的 Cobalt 体系结构中）之外，Binder 也被其他一些操作系统兼容并蓄，这其中当然也包括 Linux。Linux 中的 Binder 是开源的（网址：<http://openbinder.org>，虽然这个网站好像已经挂了，但网上还有一些该网站的镜像站点<sup>[1]</sup>）。最早的一批 Palm 开发人员离开了 Palm，加盟了 Android，同时也（为 Android）带来了 Binder。这伙人的头头就是 Dianne Hackborn——一个声名卓著的开发者，而且她至今仍是 Android 的主要设计师。她在 2006 年接受 OSNews 采访时，解释了 Binder 的一些基本概念<sup>[2]</sup>。

Android 中 Binder 的实现，与 OpenBinder 有很大的区别，而且更像 BeOS 中 Binder 的设计用途，是用作（系统中）所有框架的支撑点的。

## 那么，Binder 究竟是什么

Binder 是一种远程过程调用（Remote Procedure Call）机制。它允许应用间能够以程序调用的方式进行通信，而无须关心消息到底是如何发送和接收的。从应用程序（无论是客户端还是服务器）的角度来看，程序要做的无非就是调用一个方法（客户端）或者提供一个方法（服务端）而已。当客户端调用某个方法时，服务端程序中的对应方法就会被“神奇地”调用，而所有的“细节”显然都是由 Binder 来处理的。这些“细枝末节”的工作包括：

- **找到服务端进程**——在大多数情况下，客户端和服务端分别是两个不同的进程（除非是 system\_server 中的各个服务相互调用）。Binder 需要为客户端找出服务端进程，然后才能向它投递消息。如前文所述，这个“找到服务端”[也就是众所周知的“端点映射”（endpoint mapping）]的工作，从技术上讲是由 servicemanager 来完成的。但是 servicemanager 只负责维护一个服务目录，把一个接口名（interface name）映射成一个 Binder 句柄（handle）而已。而这个“句柄”却是 Binder 交给 servicemanager 的，一个谁也看不懂得标识符（identifier），只有 Binder 才知道它的“真正”含义，其中记录了要找的服务端进程的 PID。
- **传递消息**——如前文所述，生成获取被调用方法的参数，并将其序列化（serialize）（即把它们顺序打包到内存里的一个结构体中去）或是解序列化（deserialize）（即把结构体中各个参数一一还原出来）的代码的任务是由 AIDL 来完成的。但是从一个进程向另一个进程传递序列化了的结构体的工作，则是由 Binder 亲自完成的。客户端进程会用



BINDER\_WRITE\_READ 参数调用 `ioctl(2)`。这将会通过 Binder 发送消息，并且阻塞掉客户端进程，直到服务端进程返回结果为止（因此，代码是先写，后读）。

- **传递对象**——Binder 也可以用来传递对象。如前文所述，`service` 处理的是一种类型的对象，这些对象也包括“文件描述符”（比如 UNIX Domain socket）。传递文件描述符是一个非常重要的特性。因为这使得可信进程（比如 `system_server`）可以用原生（native）代码为一个不可信进程（比如用户安装的 App）打开某个设备或 socket。当然，这里我们假设这个不可信进程是拥有相应权限的（即在 App 的 manifest 文件中声明了该种权限）。
- **支持安全认证**——进程间通信的安全性，自然是极为重要的，消息的接受者应该能够验证消息的发送者的身份，以免落入圈套，进而殃及整个系统的安全性。Binder 可以获取到它的使用者的安全证书（PID 和 UID）并把它们安全地嵌入到消息中去。这样，服务端进程就能按合理的安全级的要求做出相应的安全认证操作。

## 使用 Binder

Binder 在所有的应用中都有使用，无论开发者自己有没有意识到这一点。在进行 Binder 操作时，所涉及的代码无非分为三个层次，如图 6-3 所示。

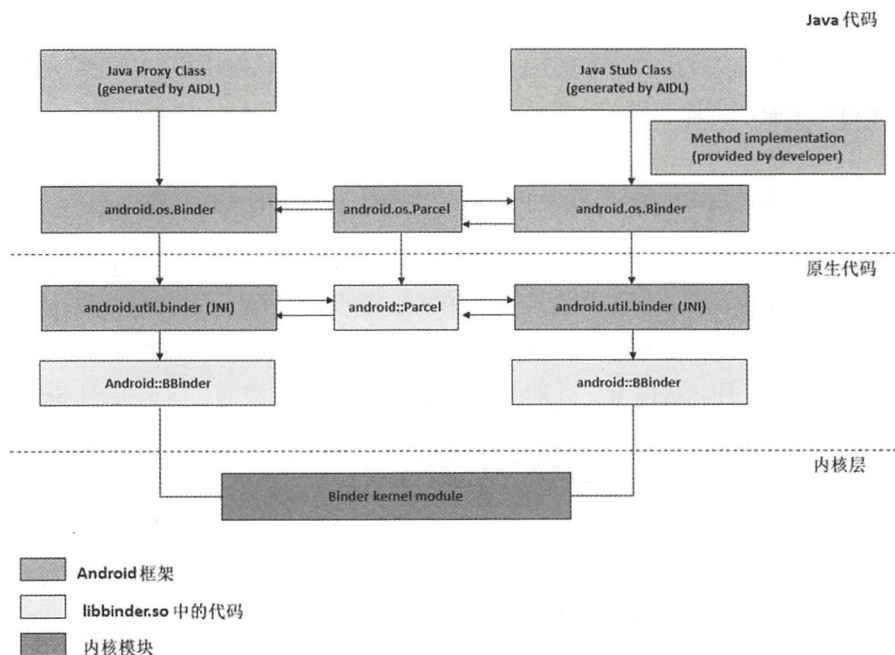


图 6-3 Binder 在客户端和服务端间传递消息时的代码执行流程

对于高级用户来说，了解上述这些信息已经足够用了，图 6-3 中总结了到目前为止我们所讲的内容。关于这些层中相关的细节——从 Java 对象到 AIDL，再到原生代码和内核中的模块，我把它们留在了第 2 本中。

## 分析 Binder 的当前使用情况

`/dev/binder` 这个文件描述符连接着任意数量的服务端。这也就意味着进程在打开这个文件描述符时，并不会去管它连接的是一个服务还是多个服务。事实上，进程甚至可以在它根本就（还）没连上任何服务时就打开它。

那么问题来了：好像也没有什么简单的方法能够让我们准确地知道指定的某个句柄所表示的服务正在被谁使用着，如果 Binder 的调试功能被启用了，我们就能用（本书官网上提供的）`bindump` 工具，根据从 Linux 的 `debugfs` 伪文件系统（`/sys/kernel/debug/binder`）那里获取的信息，分析出谁正连着什么服务。如下面这个实验所示。

### 实验：使用 `bindump` 工具，观察已打开的 binder 句柄

你从本书官网下载 `bindump` 这个工具，只是简单地扩展了 `service` 命令的一些功能而已。它会先去打开指定系统服务的一个句柄（方法类似于执行“`service check 服务名`”这样一条命令），然后再查看该进程在 `/sys/kernel/debug/binder/proc` 目录中对应伪文件中记录的内容（以获取该服务的 `node ref` 号）。每个使用 binder 的进程（在这个目录中）都有一个（对应的）伪文件，其中记录了（该进程）当前使用 binder 的实时情况。文件中 `node` 开头的各行记录中，记录的即使所使用的服务的 `node ref` 号，如果能在某条记录中找到指定服务的 `node ref` 号，那么该文件的文件名就是使用该服务的进程的 PID。因为所有 binder 调试数据都是全局可读的，所以你甚至可以在一台还没有 root 的移动设备上运行这个工具（见输出结果 6-3）。

```
#
# Inquire about wallpaper service
shell@htc_m8wl:/ $ /data/local/tmp/bindump wallpaper
Service: wallpaper node ref: 2034
User: PID 1377      com.htc.launcher
User: PID 1194      com.android.systemui
Owner: PID 1008      system_server
User: PID 368       /system/bin/service manager
#
# Who owns the batterypropreg service?
shell@htc_m8wl:/ $ /data/local/tmp/bindump owner batterypropreg
Service: batterypropreg node ref: 105785
Owner: PID 8153      /sbin/healthd
```

输出结果 6-3 使用 `bindump` 工具查看指定服务的使用者和提供者

本书的官网上还提供了一个特殊版本的 `strace()`<sup>1</sup>。这是一个能够跟踪记录（trace）Linux 系统调用的工具，这个工具有一个参数，能够使程序在运行过程中一并解析 Binder 消息（即解码 `ioctl(2)` 的操作码和载荷）。

## 6.4 system\_server

Android 设备中有几十个服务，要是再加上厂商和用户安装的 App，服务的数量或许就上百了。所幸，大部分框架服务十分的简单，并不需要一个专门的进程，只需要以线程的形式承载就足够了。不过不管怎么说，这些线程还是需要运行在一个宿主进程中的，而 `system_server` 就是这个宿主进程的提供者。

`system_server` 与 Windows 系统中的 `svchost.exe` 类似，二者提供的都只不过是一个空壳，也就是个容器进程罢了。二者的主要区别在于：`svchost.exe` 加载的是动态链接库（DLL），而 `system_server` 加载的则是 Java 类。不过，在 Android 中这样做甚至带来了一个更为重要的好处：尽管 Dalvik 虚拟机已经为共享内存做了大量的优化，但是在由同一个虚拟机可执行文件启动的另一个进程中运行各个服务，更有利于保护各类资源。但这一做法也并非没有任何风险——其中任何一个服务中出现错误，都会波及（运行在 `system_server` 中的）其他服务。不过在大多数情况下，这个问题并不算是太大。因为能运行在 `system_server` 中的只有 Android 系统的服务，厂商提供的或者其他什么服务根本就是进不来的。

`system_server` 并不是一个原生应用——它是用 Java 编写的，只有在必须调用原生代码时，才使用了一些 JNI 函数。而它加载的那些服务也同样是用 Java 编写的——尽管许多服务也会使用 JNI 跳出虚拟机直接与硬件组件进行交互。在根据 `/init.rc` 中的定义启动 Zygote 时（详见图 5-22），Zygote 会用参数 `--start-system-server`，`fork()` 出 `system_server` 进程来。`--start-system-server` 这个参数会让 Zygote 调用 `startSystemServer()` 函数，并传给它事先写好的参数——权能、组成员（`--setgroups`）、便于阅读的名字（`system_server`）以及要加载的类（`com.android.server.SystemServer`）。尽管 `system_server` 并不会以 root 权限执行，不过那也差不多了——`uid:gid` 为 `system:system`、增强的权能以及作为一大堆拥有较高权限的服务的宿主进程。从安全角度对 `system_server` 的讨论——GID 及其权能，将在第 8 章中进行。

### 启动及执行流程

作为系统中如此重要的一个支点，`system_server` 的执行流程却是相当简单的，在从 Zygote

---

1 原文如此，但实际上这个 `strace` 是一个程序。——译者注



那里被 `fork()` 出来之后，这个子进程会把自己的权限降低到前文已述的权限上。接着，它会去加载 `com.android.server.SystemServer` 类，在启动各个框架服务之前，这个类的 `main()` 方法会去完成基本的初始化工作（值得注意的是，它会提升它的虚拟机限制，并加载 `libandroid_servers.so` 执行 JNI 组件的初始化）。在所有的服务都创建完毕（及其对应的线程启动）之后，除了进入一个循环之外，`main` 线程就不需要再做什么事了。（我们希望）这个循环是个无限循环（否则系统就得停工），高层视角中的执行流程如图 6-4 所示。

不过，由于要启动大量的系统服务，`system_server` 还是需要一个一个地实例化它们。Android L 版中对这一步的流程做了大幅改进。尽管仍有大量的工作要做，但通过归类类型相似的服务，执行的流程相比之前的版本却是大大地简化了。当前，所有的服务都可以被归入以下三个大“类”中。

- **引导服务 (Bootstrap services)**：这类服务包括 `Installer`、`ActivityManagerService`、`PowerManagerService`、`DisplayManagerService`、`PackageManagerService` 和 `UserManagerService`，另外还会检测一下设备的 `/data` 分区是不是已经被加密了，或是正在被加密的过程中——因为这会对那些被指定为“core App”的 App 的启动造成一定的影响。
- **核心服务 (Core services)**：这类服务包括 `LightsService`、`BatteryService`、`UsageStatsService` 和 `WebViewUpdateService`。最后这个服务是 L 版中新增的服务，它会定期检查浏览器组件是否有更新。
- **其他服务**：基本上，这个大类里涵盖的就是剩下的所有服务。在这个类中有数十个服务（就像源码的注释中承认的那样：这是一个容纳所有有待重新归类和组织的服务的超级大口袋）。

并非所有的服务都能被应用看见。有些服务，比如 `installer` 是内部服务，因此它们就不会出现在 `service list` 命令的执行结果中，App 当然也看不见它们。我们将在下一章中逐个地讨论所有的服务，无论它是个内部服务，还是个面向 App 的服务。

在启动了所有的服务之后，`SystemServer` 的主线程就没事可干了。这时，这个线程会进入一个我们希望永远不停的循环中。之所以说“我们希望”是因为这个循环可不能被退出，否则就会抛出一个运行时异常出来。在进程内部，这个循环中的代码会不断地轮询它的文件描述符（特别是它的 `Binder` 句柄）以获取输入的消息。当有消息到达时，这些消息就会被分发给它们各自对应的目标服务予以处理。

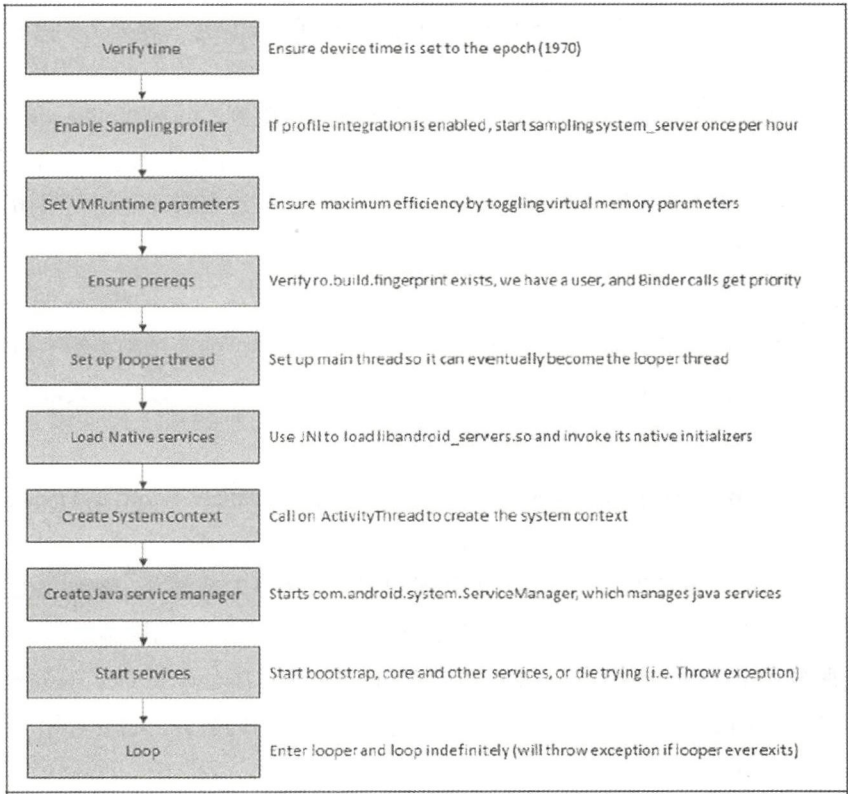


图 6-4 `system_server` 的执行流程

## 修改启动时的行为

我们可以通过设置一些系统属性的值来修改 `system_server` 的执行流程以及它启动的服务类型。

一个重要的参数是系统属性 `ro.factorytest` 的值，用它可以设置设备是不是要被配置为“工厂测试”（factory test）模式，这个系统属性的不同取值（见表 6-5）会影响到 `SystemService` 的启动流程。

表 6-5 `ro.factorytest` 的取值范围及对启动流程的影响

属性取值	对应的宏定义	含 义
0 (default)	FACTORY_TEST_NONE	正常启动
1	FACTORY_TEST_HIGH_LEVEL	不启动 bluetooth、input、accessibility、lock setting 等服务
2	FACTORY_TEST_LOW_LEVEL	用 Uid 设为 0，以便进行 App 的工厂测试

另一个重要的参数是 `ro.headless` 系统属性。如果这个系统属性的值被设为了 1，那么 Wallpaper（壁纸）服务和 System UI 服务都会被禁用。另外还有一组 `config` 系统属性，可以用来选择性地禁用一些子系统，如表 6-6 所示。

表 6-6 各个 config 系统属性影响的系统服务

config 属性名	禁用的服务
<code>disable_storage</code>	MountService
<code>disable_media</code>	AudioService, WiredAccessoryManager, CommonTimeManagementService
<code>disable_bluetooth</code>	BluetoothManagerService
<code>disable_telephony</code>	未使用
<code>disable_location</code>	LocationManagerService, CountryDetectorService
<code>disable_systemui</code>	StatusBarManagerService
<code>disable_noncore</code>	UpdateLockService, LockSettingsService, TextServicesManager, SearchManagerService, WallpaperManagerService, DockObserver, UsbService
<code>disable_network</code>	NetworkStatsService, NetworkPolicyManagerService, WifiP2pService, WifiService, ConnectivityService, NsdService, NetworkTimeUpdateService, CertBlacklist

感谢 Linux 的 `/proc` 伪文件系统，你可以通过它来检视 `system_server` 及其中的许多线程。查看它的正在使用的文件描述符实际上会是劳而无功的，因为不可能告诉你哪个文件描述符是属于哪个线程的。同理，查看进程使用的 `socket` 和 `pipe`（管道）也没什么意思。不过把这个进程中的所有线程都一一枚举出来却是非常有用的。如下面这个实验所示。

实验：阐释 `system_server` 中的各个线程

在创建 Dalvik 的线程对象时，我们可以给它命名。给线程命名需要调用底层的 `prctl(2)` 系统调用——这是一个并不广为人知，但却非常重要的 API，它可以让你在内核级别上对线程和进程进行重命名操作。在命名之后，线程的名称就可以通过查看 `/proc` 伪文件系统中该线程对应的目录里的 `status` 文件看到了。尽管这一方法并非尽善尽美，名称的长度被限制在 16 个字符之内，但是比起翻看随机生成的线程 `id`，硬猜这个线程是做什么的来说，线程能有个名字实在是方便太多了。

使用一个基本的脚本（该脚本甚至能在 Android 对 `shell` 脚本极为有限的支持下运行），你可以轻而易举地枚举出进程中所有的线程，并得到各个线程的名称（这个脚本可以对任意一个进程执行。因为它只是去遍历进程目录中的 `task/` 子目录，在这个子目录中，进程中的每个线程都有一个对应的文件夹）。在输出结果 6-4 中，`Binde` 线程及线程池（`thread pool`）被有意忽略掉了。



```

root@flounder:/proc/507/task # for t in `cat /proc/507/task/*`; do echo Thread $t `grep Name: $t/status`: done
507: Name: system_server # The main thread (same as PID)
512: Name: Heap thread pool # L: ART Heap thread pool
514: Name: Signal Catcher # Dalvik signal catcher
515: Name: ReferenceQueueD # Dalvik Reference Queue Daemon
516: Name: FinalizerDaemon # Dalvik object finalizer
517: Name: FinalizerWatchdog # Dalvik finalizer watchdog
518: Name: HeapTrimmerDaemon # L: ART Heap Trimmer Daemon
519: Name: GCDAemon # Garbage Collector (L: "GCDAemon", for ART)
524: Name: SensorService
525: Name: SensorEventHookR
526: Name: android.bg
527: Name: ActivityManager
529: Name: FileObserver # FileObserver$Thread
530: Name: android.fg
531: Name: android.ui
532: Name: android.io
533: Name: android.display
534: Name: CpuTracker # Created by ActivityManager
535: Name: PowerManagerService # Created by PowerManagerService
537: Name: BatteryStats_wa
562: Name: PackageManager # Created by
594: Name: PackageInstaller # PackageManager
596: Name: AlarmManager # Created by AlarmManagerService
597: Name: InputDispatcher # Started by InputManager
598: Name: InputReader # Started by InputManager
599: Name: MountService # Created by MountService
600: Name: VoldConnector # Created by MountService
602: Name: NetdConnector # Created by ConnectivityManager
603: Name: NetworkStats
604: Name: NetworkPolicy
605: Name: WifiP2pService
606: Name: WifiStateMachine
607: Name: WifiService
608: Name: ConnectivityService # Created by ConnectivityManager
609: Name: NsdService # Neighbor Services Discovery (State Machine Thread)
610: Name: mDnsConnector # Created by NsdService
611: Name: ranker # Created by NotificationManagerService
613: Name: AudioService # Created by AudioService$AudioSystemThread
622: Name: UEventObserver # Kernel uevent observer (shared by many services)
623: Name: backup # Created by BackupManagerService
626: Name: WifiWatchdogSta
627: Name: WifiManager
628: Name: WifiScanningService
629: Name: WifiRttService
630: Name: EthernetService
634: Name: LazyTaskWriterThread # ActivityManager's TaskPersister
635: Name: UsbService host
844: Name: watchdog
845: Name: SoundPool # AudioService$SoundPoolListenerThread
846: Name: SoundPoolThread # AudioService$SoundPoolListenerThread
906: Name: NetworkTimeUpdate # NetworkTimeUpdateService's HandlerThread
984: Name: IPC Thread
1009: Name: WifiMonitor
1507: Name: SynoHandler-0
1513: Name: UsbDebuggingManager

```

输出结果 6-4 遍历进程中的各个线程

一般而言，线程 id (TID) 是不可预测的，但是由于 system\_server 中大部分线程是紧邻着启动的，所以它们的 TID 也大致是顺序增长的，所以通过观察这些 TID 的具体数值，也能让你对系统中框架服务的启动顺序有个大致的印象。

本章小结

本章讨论了 Android 框架服务的架构，解释了 Android 中通过远程过程调用（RPC，Remote Procedure Call）进行进程间通信（IPC，Inter Process Communication）的底层机制，重点讲述了 servicemanager 和 service 工具。接下来，本章重点介绍了 system\_server 进程，它是作为 Android 中大量框架的宿主进程而存在的，完全使用 Java 编写。

这当然会引出更详细的讨论的需求，特别是对大量各种不同类型的服务，以及 Binder [它是帮助实现远程过程调用（RPC）的传输机制] 的讨论。不过这些讨论还是要留待第 2 本中进行。

本章讨论涉及的文件

组 件	文 件	内 容
ServiceManager	f/native/cmds/servicemanager/service_manager.c	service manager 的源码
	frameworks/native/cmds/servicemanager/binder.[ch]	Binder 的接口
SystemServer	f/b/services/java/com/android/server/SystemServer.java	SystemServer 类

参考文献

[1] OpenBinder documentation (mirror):  
<http://www.angryredplanet.com/~hackbod/openbinder/docs/html/>

[2] OSNews, Interview with Dianne Hackborn:  
<http://www.osnews.com/story/13674/>

## 第 7 章

# 从 Linux 角度看 Android

Android 开发人员习惯于用我们上一章中讨论的 Android 软件运行生命周期中的相关概念去看他们的软件。不过从 Linux 角度看，Android 应用也是 Linux 进程，它们和系统中的其他进程本就没多大的区别。

这一章里，我们将着重从这一角度展开讨论。我们首先讨论的是 `/proc` 伪文件系统——Linux 系统中可以通过它监视和跟踪进程，在第 2 章中对它的介绍过于简略，在这一章中我们将还它以应有的篇幅。我们会讨论表示各个进程/线程的 `/proc/pid` 目录中的各个文件和子目录——通过它们，我们可以实时地轮询到进程/线程的当前实时状态信息。我们先会去讨论几个反映进程工作目录的符号链接，然后我们去重点关注一下非常有用的 `fd/` 和 `fdinfo/` 子目录——其中记录了准确而详细的该进程打开的文件描述符的属性信息。接下来讨论的是 `status` 文件，这个文件中记录了大量进程级的（特别是线程状态和虚拟内存的）实时使用信息。

在进行性能诊断时，虚拟内存是个很重要的方面。所以接下来的一节将着重介绍用户态内存管理的基本知识。我们不光介绍相关概念的理论知识，还会联系 `/proc` 伪文件系统中的 `smaps` 文件，具体给出这些概念在实践中的应用。此外，我们还会介绍两个工具 `procrank` 和 `librank`，我们可以使用这两个工具获得大量内存实时使用情况的信息。我们还会说明当发生了令人头皮发麻的内存不足（Out-Of-Memory）的情况时 Android 的应对方案，以及它对 Android App 生命周期造成的严重影响——这个应对方案使 App 一直被笼罩在朝不保夕、不知道什么时候就会被干掉的阴影下。

最后，我们要去讲一下系统调用。我们将向你演示 `toolbox` 中的 `ps` 工具、`profs` 中的 `wchan` 和 `syscall` 文件以及全能的 `strace` 工具 [使用 `strobe`，可以实时地跟踪（trace）进程的行为]。

要读懂这一章的内容，你必须理解操作系统理论中的一些概念，我们会通过大量动手实验进一步讲解这些必须掌握的概念。本章中所演示的方法和实验都基于 Linux 内核特性，这使得



它们既适用于 Android 又适用于 Linux 系统。因此你也可以认为本章的内容是关于 Linux 系统调试这一相关专著实在少得令人吃惊的主题的。

## 7.1 重温/proc

对于/proc 伪文件系统，我们在第2章里已经接触过了。不过那时只不过是蜻蜓点水式的一瞥，远远没有揭示/proc 伪文件系统的极端重要性。事实上，每个进程（和线程）在/proc 中都有对应的目录，其中记录了大量关于程序内部运行状况的实时状态信息。

为了更直观地理解各个进程目录中记录的信息，我们不妨暂且借用面向对象的概念来看待“进程”：我们可以把每个进程都视为一个“process”类的实例，所有这些实例都有相同的属性——尽管各个属性的属性值很可能是不一样的。在/proc 伪文件系统中，各个进程对应目录中的伪文件的作用，只不过是把这些属性值显示给你看而已——在某些情况下，也就是某些伪文件可以写的情况下，你甚至还可以修改这些属性的值。

输出结果 7-1 中显示的是当前正在使用的 Android shell 自身在/proc 中对应目录里的子目录和文件（请注意，我们是用符号“\$\$”而不是/proc/self 来表示当前 shell 的进程 ID 的——因为如果用的是/proc/self 的话，显示的将是 ls 进程，而不是当前 shell 在/proc 中的对应目录）。另外，你并不一定要用 cut 语句调整 ls 命令的输出，我这样做只是为了让输出结果更好看些罢了。

```
shell@flounder: /data$ ls -l /proc/$$ | cut -c1-10.55-
dr-xr-xr-x attr          # Attr
-r----- auxv           # ELF AUXiliary Vectors, in binary form
-r--r--r-- cgroup        # Control Group membership of process
-rw----- clear_refs     # Clear page referenced bits in smaps
-r--r--r-- cmdline       # Command line (argv[]), NUL separated
-rw-r--r-- comm          # 16-byte argv[0]
lrwxrwxrwx cwd -> /data   # Process current working directory, as symlink
-r----- environ        # Environment (as per set or getenv()), NUL separated
lrwxrwxrwx exe -> /system/bin/sh # Full path of executable, as symlink
dr-x----- fd           # Directory containing open file descriptors, as symlinks
dr-x----- fdinfo       # Directory containing metadata on open file descriptors
-r--r--r-- limits        # Process hard/soft limits, as per ulimit or setrlimit(2)
-r--r--r-- maps          # Process address space memory map
-rw----- mem           # Process virtual memory, as a pseudo-file
-r--r--r-- mountinfo      # Mounted file systems, as process sees them (mount namespace)
-r--r--r-- mounts         # Mounted file systems, slightly different format
-r----- mountstate      # Mounted file systems, slightly different format
dr-xr-xr-x net           # Network statistics of process network namespace
dr-x-x-x-x ns            # Namespaces of this process, as symlinks (commonly, mnt)
-rw-r--r-- oom_adj        # Out-Of-Memory (old) score adjustment. Used by ActivityManager
-r--r--r-- oom_score      # Out-Of-Memory score. Determines process killability on OOM
-rw-r--r-- oom_score_adj  # Out-Of-Memory (new) score adjustment. Used by ActivityManager
-r--r--r-- pagemap        # Page map of process
-r--r--r-- personality    # OS Personality. Commonly, this is 00000000 (Linux)
lrwxrwxrwx root -> /      # Process root directory. Always symlinks to /, unless chroot(2)ed
-rw-r--r-- sched          # CFS statistics for this process, in human-readable form
-r--r--r-- schedstat      # More CFS statistics for this process
-r--r--r-- smaps          # q.v maps, but with more detailed information per memory region
-r--r--r-- stack          # kernel stack of process (technically, main thread)
-r--r--r-- stat           # Statistics (from task_struct), in machine-readable form
-r--r--r-- statm          # More statistics, in machine-readable form
-r--r--r-- status         # Statistics (from task_struct), in human-readable form
-r--r--r-- syscall        # Current system call and arguments
dr-xr-xr-x task          # Subdirectory containing entries for process threads
-r--r--r-- wchan          # Wait Channel in kernel
```

输出结果 7-1 Android L 系统（3.10 内核）的/proc 伪文件系统中各个表示对应进程的目录中存放的内容

记住——这里没有一个是真正的文件。这也就意味着两件事：

- 在内核版本不同的系统中，这张表中列出的内容可能稍有不同。一般而言，越新的内核里，你能看到的伪文件也就越多——尽管在编译内核时可以关掉对其中一些功能（伪文件）的支持。
- 这些文件并不存在，只有在你请求访问它们时，它们才会临时出现一下。这也就意味着每次你看到的文件内容都可能是不同的。在使用 `ls` 命令查看这些目录中的内容时，内核甚至都不会费心去查询文件的大小——这也就是如果你不像输出结果 7-1 中那样用 `cut` 命令调整输出结果的话，你会看到所有文件的大小都是 0 的原因。

后一点是非常值得思考的：由于所有的文件完全都是虚拟出来的，所以维护 `/proc` 中的文件和目录是没有任何开销的。内核只需在正常操作的过程中记录相关信息即可，所有这些文件和目录全都是虚假的——只有当用户请求访问它们时，内核才会实时地收集相关信息，并以伪文件系统的方式向用户提供这些信息。这使得 `/proc` 文件系统成了一种 `trace` 系统或程序的异常强大的机制，而使用它的方式通常是轮询（`polling`）。

以轮询的方式进行 `trace` 意味着 `trace` 程序或脚本必须保持每隔一段时间就要访问 `/proc` 中的指定文件或目录一下。这是因为 `/proc` 中的文件或目录并没有在相关数据发生改变时调用回调函数的功能（至少目前还不支持这一功能）。这一做法有一些缺点：如果轮询的时间间隔太长，你可能错过某些你想要截获的事件。但它的优点（即零开销）显然远远大于它的缺点。此外，伪文件中的信息是以方便人工读取和易于解析的格式给出的，这也是另一个显而易见的好处——我们将在这一章中共同见证这一点。

由于内核更新的频率实在是太快了，这使得（Linux 系统中自带的）文档的内容都有可能过时——更何况也不是所有的东西都有相应的文档说明。接下来，我们会重点讨论一些比较重要的伪文件——在分析系统性能和调试时，它们用起来非常顺手。

## 符号链接：`cwd`、`exe` 和 `root`

仔细看一下输出结果 7-1，有三个东西 `cwd`、`exe` 和 `root` 显得那么与众不同。因为其他东西都是伪文件，只有这三个东西是符号链接。

之所以把这三者用符号链接表示主要是为了图个方便。这三个符号链接指向的都是真实存在的文件或目录，对于用户来说，可以很方便地通过符号链接直接操作链接所指向的文件或目录（比如 `cat(1)`，`ls(1)`）。如果不用符号链接的话，就不得不把目标文件或目录的路径放在某个伪文件中，让用户先把这个路径读取出来，然后再把它放在要执行的命令的参数里，才能操作

相关文件或目录。

这三个符号链接向你提供的是进程中最重要的高层（high-level）状态信息。

- **cwd**: 它指向的是进程当前的工作目录。我们可以从输出结果 7-1 里的提示信息中得知，shell 的当前工作目录是/data——这也正是符号链接 **cwd** 所指向的地方。我们可以做一个有趣的实验：用 **cd** 命令，切换到任意一个目录上，然后再执行 **ls -l /proc/\$\$/cwd**。你可以看到，尽管你能够不断地随意切换目录，但你就是隐藏不了当前工作目录——无论你什么时候去查询 **cwd** 符号链接，内核总是会把它指向你查询时的当前工作目录，这也就是说，你总能得到正确的目录。
- **exe**: 它指向的是用来产生这个进程的可执行文件（也就是由系统调用 **execve(2)** 加载的那个可执行文件）的全路径。这个东西很有用，因为许多进程会修改自己的进程名，让 **ps** 显示出修改后的进程名，但它们改不了这个符号链接。
- **root**: 它指向的是根（root）目录。通常情况下，它指向的就是真正的根目录“/”，但是如果某个进程中调用了 **chroot(2)** 函数，把某个子目录定义为它新的根目录，看一眼这个符号链接就能很明显地发现这一操作。

**fuser** 工具的作用是找出系统中所有打开了指定目录中文件或子目录的进程；**lsdf** 工具的作用是按进程 ID 逐个列出各个进程打开的文件。这两个工具里就会用到 **cwd** 和 **root** 这两个符号链接。知道了这一点，我们就可以轻而易举地写一个 shell 脚本来实现这两个工具的功能。在没有预装这两个工具的系统中，这一招将是非常有用的。在下面这个实验里，我们用看上去并不怎么有用的 **exe** 符号链接向你展示一下类似的 shell 脚本编制技巧。

## 实验：区分 32 位/64 位的 Android 应用

乍一看，**exe** 这个符号链接似乎没什么大用——毕竟在绝大多数情况下，可执行文件都不会真的在运行过程中修改自己的名称……不是吗？

还真不是！比如在 Android 系统中，各种跑在 Dalvik 虚拟机里的应用，它们从 Zygote 进程 fork 出来之后，会把它们的进程名改成它们正在执行的主类名。修改进程名是通过调用系统调用 **prctl (PR\_SET\_NAME...)**，把 **argv[0]** 设为相应的字符串的方式完成的。所有应用真正的进程名应该是 **/system/bin/app\_process**，这才是加载这些应用的“真正的”虚拟机实例。在 Android L 系统中，这一点甚至更有用，因为 32 位的应用显然对应的是 **app\_process32**；而 64 位的应用对应的则是 **app\_process64**。下面这个实验将会向你展示如何把这一点用在一个 shell 脚本上。



```

root@flounder:/# cd /proc; for p in [0-9]*; do
> if ls -l $p/exe | grep app_process32 > /dev/null; then # isolate 32-bit, but ignore output
> echo "cat $p/cmdline" \ (PID $p) is a 32-bit app
> fi
> if ls -l $p/exe | grep app_process64 > /dev/null; then # isolate 64-bit, but ignore output
> echo "cat $p/cmdline" \ (PID $p) is a 64-bit app
> fi
> done
com.google.process.location (PID 10446) is a 64-bit app
com.google.android.inputmethod.latin (PID 10702) is a 64-bit app
... etc.. etc..
com.google.android.apps.maps (PID 14610) is a 32-bit app
com.google.android.talk (PID 15219) is a 32-bit app
zygote64 (PID 211) is a 64-bit app
zygote (PID 212) is a 32-bit app
system_server (PID 511) is a 64-bit app
com.android.systemui (PID 691) is a 64-bit app
com.android.server.telecom (PID 930) is a 64-bit app
com.android.phone (PID 988) is a 64-bit app

```

输出结果 7-2 利用 /proc 中的符号链接 exe，辨别应用是 32 位的还是 64 位的

为了更好地理解这个脚本，请注意编写这个脚本时，构造循环遍历各个进程对应目录的方法和使用 grep 自动分析文件内容的技巧。

1. 切换到 /proc 目录：这一章中的所有分析都是从这里开始的。

2. 用正则表达式 `[0-9]*` 筛出并逐个遍历那些以数字开头的子目录：在 /proc 目录中，除了表示各个进程的子目录之外，还有一些其他的文件和目录。我们只关心那些表示进程的子目录，所以我们要挑出这些以数字开头的子目录。在下面的循环中，将以 \$p 来表示这些 PID（也就是子目录的目录名）。

3. 检查 /proc 目录中的各个 exe 符号链接：注意 grep 的使用方式，它的输出结果会被直接丢弃掉（> /dev/null），我们只关心有还是没有输出——if 语句只需要 grep 的这个大致执行结果，就能确定应该跳转到哪里去执行。由于实际上只会有三种不同的情况（exe 符号链接指向的是 app\_process32、exe 符号链接指向的是 app\_process64 和 exe 符号链接既不指向 app\_process32 也不指向 app\_process64），所以我们没有使用 if-else 结构，而是用了两个独立的 if 语句。

4. 向用户输出结果：我们在其中使用了 cmdline 文件，这个文件中存储的是进程启动时输出给它的各个命令行参数（以 NULL 符号分割），我们用 cat(1)，只输出第一个参数（argv[0]，也就是程序的完整路径）。用这一招比使用 comm 文件要好，因为 comm 文件中存储的程序名至多 16 个字节，超出的话会被截短。另外还请注意 \$p 的用法，这个 \$p 是循环变量，表示的是 PID（这也是我们一开始就先用 cd(1) 切换至 /proc 打下的伏笔）。

乍一看，在这里编写一个 inline 的 shell 脚本或许会把一些读者给吓跑。特别是考虑到 shell 脚本晦涩难懂的语法，这并不难理解（这也是为什么在这里要设计这样一个实验的原因，我强烈建议你至少亲自动手做一遍这个实验）。记住，每个像这个脚本一样的 inline 脚本都可以很方便地存到一个文件中去，然后你再用 chmod 给它加上“+x”权限——这样你的工具箱里就又多了一件新兵器。我们可以直接套用这个实验中给出的脚本，遍历所有表示进程的子目录，并用

grep(1)分析/proc 伪文件系统中需要解析的文件或文件夹中的内容，这样就能拿出一个专为你的需求而编写的，而且还可以重用的工具出来，并且不论是在 Android 系统还是 Linux 系统中，它都能正常使用。

## fd

进程得通过文件描述符（file descriptor）才能完成 I/O 操作。不论是用哪种语言——Java、C 还是其他什么语言，被打开的文件、管道（pipe）、套接字（socket）统统都会映射为数字形式表示的文件描述符。进程创建时会默认打开三个文件描述符，它们是标准输入（stdin）、标准输出（stdout）和标准错误信息（stderr）——它们各自对应的文件描述符分别是 0、1 和 2。当进程打开或创建一个文件（或者套接字、管道……）时，系统就会让下一个可用的描述符指向这个新创建出来的对象。至于进程看到的这个数字（文件描述符），事实上只是指向内核中某个对象的一个句柄（handle）而已——在内核中，每个进程都有一个数组，其中记录着所有该进程打开的文件对象（包括套接字、管道等），而这个句柄就是对应文件在该数组中的索引号。

因此，最重要的是要能实时地给出给定进程当前正在使用哪些描述符。这样的工具确实已经有了一些——特别是 lsof（list of file），它会把每个进程打开的所有文件以及其他一些信息一起列出来。不过除了总是依赖 lsof（因为不是每个 Linux 发行版本中都预装了 lsof）之外，学会直接去 lsof 获取的信息源头，即/proc/pid/fd 那里获取信息也是很重要的。

fd/目录里存放的是和我们上一节中讨论的 cwd、exe、root 一样的符号链接。所以只要在给定 PID 的 fd/目录上执行 ls -l 命令，就能看到该进程正在使用那些文件，如输出的结果 7-3 所示。这一招极其有用，君不见，我们在之前章节中甚至已经演示过这一技巧好几次了。这也是我知道的简单的方法了，不过，必须要注意到：如果你不是该进程的所有者（owner）的话，在去列文件描述符之前，你必须先有 root 权限（不然，你连/proc/pid/fd/目录都进不去）。

```
root@flounder:/proc/151/fd # ls -l
lrwx----- root    root    2015-01-04 10:19 0 -> /dev/null
lrwx----- root    root    2015-01-04 10:19 1 -> /dev/null
lrwx----- root    root    2015-01-04 10:19 10 -> socket:[11955]
lrwx----- root    root    2015-01-04 10:19 11 -> socket:[6639]
lr-x----- root    root    2015-01-04 10:19 12 -> /dev/alarm
lrwx----- root    root    2015-01-04 10:19 13 -> socket:[11986]
lrwx----- root    root    2015-01-04 10:19 2 -> /dev/null
lrwx----- root    root    2015-01-04 10:19 3 -> socket:[10409]
l-wx----- root    root    2015-01-04 10:19 4 -> /sys/kernel/debug/tracing/trace_marker
lr-x----- root    root    2015-01-04 10:19 5 -> /system/framework/framework.jar
lr-x----- root    root    2015-01-04 10:19 6 -> /system/framework/core-libart.jar
lrwx----- root    root    2015-01-04 10:19 7 -> socket:[11323]
lr-x----- root    root    2015-01-04 10:19 8 -> /system/framework/framework-res.apk
lr-x----- root    root    2015-01-04 10:19 9 -> /dev/__properties__
```

输出结果 7-3 通过/proc/pid/fd，列出某一进程（Zygote）的文件描述符



要是遇到的都是正常的文件，这就把所有文件名都给列出来了。但是如果遇到了套接字（socket）的话，就另当别论了。因为在文件系统中没有用来表示 socket 的文件（尽管有些 UNIX domain socket 是有对应的文件的），所以 socket 是没有相应的符号链接的。尽管可以在这里放上一个伪符号链接，在其中记上当前这个 socket 的 IP 或者域名也是很方便的，但截至本书写作之日，Linux 内核作者选用的做法还是给出 socket 对应的 inode 的编号，而不是我刚才说的这个最省事的方法。在上面这个输出结果中，我们也可以看到，文件描述符 7、10、11 和 13 对应的 socket 就是以这种方式给出的。问题是这些 socket 又是连到哪里去的呢？

幸好，在 /proc 伪文件系统中还有其他一些伪文件能够帮你解决这一问题。在下面这个实验里，就会教你怎么得到 socket 真实指向——不管它是 UNIX domain socket 还是基于 IP 的 socket。

### 实验：通过 /proc/net 找出 socket 的真实指向

尽管只要用 lsof(1) 这类工具（不过 lsof(1) 不是 toolbox 工具箱中的工具）就能自动为你找出包括 socket 在内的所有描述符的含义。但只要再知道一些关于 /proc 中存储的数据的知识，你自己也能很快得出答案。Linux 和 Android 中的 socket 通常是下面三种类型之一<sup>1</sup>。

- **UNIX domain socket:** 这类 socket 仅用于本地通信，其中的一些是 named 的，也就是说，它们在文件系统中是有对应的文件的。不过实际上，这些并不是真正的文件，domain socket 只是内存中的内核数据结构，文件名的作用只不过是为了确保 socket 在整个系统中的唯一性。在 Android 系统中，这些 socket 和另一种 socket 一起（这种 socket 是以字符“@”开头的，由于“@”是隐藏文件的符号，所以这些文件是不会在文件系统中显示出来的）存放在 /dev/socket 目录中，除此之外，其他类型的 socket 都是 unnamed 的。domain socket 的相关实时使用情况信息，则被内核存放在 /proc/net/unix 文件中。
- **基于 IP 的 socket:** Linux（和 Android）可以同时使用 IPv4 和 IPv6 这两个不同的地址族（address family），然后我们还可以把它们再进一步细分为 udp 和 tcp 这两个不同的协议类型。因此，毫不奇怪，存放相关实时使用情况信息的文件一共有四个：tcp6、udp4、tcp 和 udp。
- **Netlink socket:** 这种 socket 被用作一种高效的内核用户空间通知机制。它只在 Linux 中被使用，而且因为它所具有的多播功能（也就是说，它能让一组用户共享同一个 socket，向这些用户同时发送通知）而受到青睐。这类 socket 的实时使用情况信息存放在 /proc/net/netlink 文件中。

---

1 这三种不过是最常见的三种，其他类型的 socket（比如，raw socket）的相关信息当然也在其他文件里。  
——原注



对于基于IP的socket来说,在`/proc/net`中的各个存放socket相关实时使用情况信息的文件<sup>1</sup>中,找到其对应的inode编号的方法很简单:因为这里只有4个文件,所以只需要用`grep(1)`就能迅速完成这一任务,具体做法如输出结果7-4所示。

```
shell@flounder:/ $ grep 471470 /proc/net/*
/proc/net/tcp6: 19: 0000000000000000FFFF00006E01000A:E0AE 0000000000000000FFFF00000E7BC2AD:01BB
00000000:00000000 00:00000000 00000000 10060 0 471450 1 0000000000000000
23 4 32 10 -1 com.google.android.apps.maps 14610 droid.apps.maps
```

输出结果 7-4 从`/proc/net`中找出给定的IP socket

“busybox netstat”或者“busybox lsof”(只可惜,这两个命令都不在toolbox中)之类的工具是可以解析上面这个输出结果的,不过你要是想手工去做解析的话,也只要根据规定的格式,把这些十六进制值和它们表示的含义一一对应起来就行了。这行数据的格式是:

```
##: Loc_v6/v4:Port Rem_v6/v4 state .. inode#... pname pid comm
```

上述这些就是你所需要的全部信息。

可惜的是,UNIX domain socket解析起来就没这么简单了。有时,在`/proc/net/unix`目录中用`grep`命令搜索时,会得到socket的名字(name),但很多时候要找的socket却是unnamed的——出现这种情况时,会使我们很难正确解析连在这个socket的另一端上的到底是什么。有时,可以试试去查看比这个inode编号略高或略低的inode编码是不是还是一个unnamed socket,或者我们可以试试“`ls -lR /proc/[0-9]*/fd`”命令——用这条命令常常能找出socket的另一端是谁。不过因为socket并不总是成对创建的,所以这些都不是百试百灵的方法。这时,还有个简单排除规则可以助你一臂之力:在内核符号表和`/proc/kcore`中找不到的inode编号,都可以去掉。

## fdinfo

乍一看,fdinfo/目录看上去不是那么重要——里面的东西看上去和fd/目录里的差不多,只不过不是符号链接罢了(所以列出的内容也不是五颜六色的)。不过事实上,fdinfo/目录中存放的信息和fd/目录中的是一样重要的。

fd/目录中的每个文件描述符在fdinfo/目录里都有一个对应的同名文件,其中记录了与该文件相关的元数据(metadata)。设备驱动和文件系统的实现代码,可以通过这个文件,将该文件描述符当前状态的相关信息,传递到用户空间。只是在实践中,这样做的很少,所以一般情况下,其中只有由内核自身维护的默认信息,其中特别值得关注的是:

<sup>1</sup> 从技术上讲,在`/proc/pid/net/family`文件中可以得到更精确的结果,因为socket必须保存在某个命名空间(namespace)中。不过`/proc/net`也提供了这样一个命名空间,所以这里的这个做法仍是有效的。——原注

- **flags**——这个 flags 就是使用 open(2) 系统调用、创建或打开文件时传入的“flags”参数。可以在头文件 <fcntl.h> 中找到相关标志位的定义。
- **pos**——也就是“文件指针”当前的位置 (position)，即文件下一个要读/写的字节所在的位置。

pos 字段里记录的是真正的宝藏。它让你能够监控某个进程，粗略地估计它大概已经运行到了什么地方——所有这些信息都是以非常易于读取的格式提供的。只要用几行 shell 脚本，你就能利用（系统提供的）这一功能，根据自己的需要，写出一个在进程运行到某一特定条件下时，才会执行指定操作的工具来。代码清单 7-1 给出的就是这样一个例子。

代码清单 7-1 监视进程并在它读写到文件的特定位置上才会执行操作的 shell 脚本

```
PID=$1                # PID is first argument
FD=$2                 # FD to watch is second argument
OFFSET=$3             # OFFSET to monitor is third argument
CUT_COMMAND='busybox cut' # needed because toolbox doesn't have cut built-in

# This isolates just the numerical offset from the fdinfo entry of $FD

# Get the data | isolate pos line | isolate numerical value
CUROFF=`cat /proc/$PID/fdinfo/$FD | grep pos | $CUT_COMMAND -d':' -f2`

if [[ $CUROFF -gt $OFF ]; then
    echo Do something
    # Insert command to execute on trigger here
else
    echo Nothing to do.
fi
```

上面这个脚本的缺点之一是：它是通过轮询 (poll) 而不是通知机制工作的。在有些情况下，在两次轮询之间，pos 字段中的结果可能会发生多次变化，这时就看你的人品了——如果具体执行轮询操作的时间不巧的话，你或许会错过进程读写到你要监视的那个文件位置上的时间点。你可以使用缩短执行这个脚本的周期或修改 OFFSET 参数、增加冗余量（也就是说，把 OFFSET 参数设在你真正要监视的偏移量之前一点，当程序运行到这个提前了的点上时，使用更保守的调试技术）等方法缓解这一问题。



上面给出的这个脚本，体现了这样一种设计思路：我们可以从 /proc 伪文件系统中的各个子目录和文件中获取信息，再根据获取到的这些数据执行相关操作。由于 /proc 伪文件系统是以伪文件的形式导出相关数据的，所以只要知道正确的文件名，并掌握 UNIX 系统中各种数据处理工具（cut、grep、sort 及其他类似的工具）的使用方法，就能根据你自己的需要，编写出各种能完成不同功能的工具来。事实上，大多数你所知道或是钟爱的（或者至少是“常用的”）工具，都可以用这一思路 [遍历 /proc 中的各个文件，尤其是与进程/线程相关的文件（特别是接下来就要讨论的 stat 和 status）]，编写脚本予以重新实现。

## status

status 文件可以算是一个一站式存储中心，其中存放着你能想到的所有与进程相关的信息——而且可能比你想要的还要多得多。因为内核所做的，实际上是把在 Linux 内核中被用作进程控制块（PCB，Process Control Block）的 task\_struct 这一庞杂的结构体，以便于人类阅读的形式 dump 在了 status 文件里。借此，我们能一窥内核在处理进程时所见的信息，如输出结果 7-5 所示。

```
shell@flounder:/proc $ cat /proc/511/status
Name: system_server # Same as /proc/511/comm
State: S (sleeping) # or R - running, T - Stopped, D - Uninterruptible (deep) sleep
Tgid: 511 # Thread Group id: The real process id
Pid: 511 # Thread, not process id
PPid: 211 # Parent Thread Group id
TracerPid: 0 # Any ptrace(2) attached process, like strace, gdb or debugger
Uid: 1000 1000 1000 1000 # Real, Effective, Set and File-System UIDs
Gid: 1000 1000 1000 1000 # Real, Effective, Set and File-System GIDs
FDSize: 2048 # Maximum # of file descriptors allowed
Groups: 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1018 1032 3001 3002 3003 3006 3007
VmPeak: 2367448 kB # Virtual Memory high-water mark
VmSize: 2258636 kB # Virtual Memory size, present
VmLck: 0 kB # Memory locked by mlock(2) APIs
VmPin: 0 kB # Pinned memory
VmHWM: 178100 kB # RSSPeak - i.e. Resident memory footprint high-water mark
VmRSS: 151600 kB # Resident memory footprint, present
VmData: 230448 kB # Size of data segment (heap memory)
VmStk: 8192 kB # Size of process thread stacks
VmExe: 16 kB # Size of executable
VmLib: 120016 kB # Memory used by shared library (.so) files
VmPTE: 968 kB # Memory used by Page Table Entries
VmSwap: 0 kB # Memory used by process in swap (If no swap, always 0)
Threads: 82 # Number of threads. If > 1, this is a multi-threaded process
SigQ: 3/6826 # Handled Signals/Size of signal queue
SigPnd: 0000000000000000 # Bitmask of pending signals (for thread)
ShdPnd: 0000000000000000 # Shared pending signals for process
SigBlk: 00000000000001204 # Bitmask of signals blocked (by SIG_BLOCK)
SigIgn: 0000000000000000 # Bitmask of signals ignored (by SIG_IGN)
SigCgt: 00000002000094f8 # Bitmask of signals caught (by handlers)
CapInh: 0000000000000000 # Bitmask of inherited capabilities
CapPrm: 000001007813c20 # Bitmask of permitted capabilities
CapEff: 000001007813c20 # Bitmask of effective capabilities
CapBnd: 0000000000000000 # Bitmask of capabilities bounding set
Cpus_allowed: 3 # Bitmask of CPUs allowed (3 = 0011)
Cpus_allowed_list: 0-1 # List of CPUs allowed, for the hex-challenged
voluntary_ctxt_switches: 40684 # Voluntary (system call induced) context switches
nonvoluntary_ctxt_switches: 13471 # Nonvoluntary (preemption induced) context switches
```

输出结果 7-5 加了注释的/proc/pid/status 文件的内容

这个输出结果相当冗长，所以把一些不是一眼就能看出其含义的字段搁置起来，暂时不予讨论还是很明智的。

别把 pid、tid、tgid 和 ppid 搞混了

你是不是已经下意识地吧“pid”当成进程 ID（Process ID）了？好吧，但可惜，它不是！最初，Linux 确实是把 PID 用作进程 ID 的，但随着新千年到来，Linux 中整合进了许多现代操



作系统的特性，开始以线程而不是进程作为处理器调度的单位了。所以，pid 实际上表示的是当前我们读取的这个伪文件的线程 ID，而非进程 ID。而进程，只是指共享了一系列资源（虚拟内存地址空间、文件描述符等）的一组线程而已，所以记录进程 ID 的其实应该是 tgid 这个字段。

可能有些读者一时还难以接受，特别是在上面这个例子中，tgid 和 pid 字段中记录的还都是同样的值。这并没有问题：因为对于某个进程的主线程（main thread）来说，pid 的值和 tgid 的值总是一样的。事实上，我们还可以利用这一点，很方便地确定某个线程是不是其所属进程的主线程——换言之，也就是它是不是这个线程组中的第一个线程。而对于其他各个子线程而言，pid 的值会变，但 tgid 的值仍然还是原来的值，不会变。

基于同样的理由，PPID 字段（父进程 ID），更准确地来说，表示的也应该是父线程组的 ID。只是 PTGID 读起来总有那么些拗口，所以它还是被称作 PPID，了解各个进程之间的父子关系是很重要的：因为父进程要负责收集子进程的返回码（也就是 main() 函数的返回值，或者调用 exit() 函数时传入的参数）。此外，当我们选择要杀掉某棵进程树时，该进程的所有子孙进程也会被一起干掉。

UNIX 系统中的大多数工具都会“选择性地”只显示主线程的相关实时信息，从这个角度讲，进程这一概念（或者说是“错觉”）还是成立的——因此在这一节中，我们仍然还会沿用这一概念。而 ps 命令，为了向后兼容早期的概念，也还是“昧着良心”把 tgid 输出为 pid。在 Linux 中，用“ps -L”命令查看各个线程时，会把 pid 显示为 LWP。任何一个线程组中的所有线程，都可以在 task/子目录中被找到。具体过程如下面这个实验所示。

### 实验：在 /proc 伪文件系统中查看各个线程和进程

对于系统管理员或者程序调试人员而言，能仔细检查那些拥有多个线程的进程中的各个线程总是很有用的。/proc 伪文件系统中提供了每个线程的实时状态信息。不像 Linux 中的 ps 提供了（BSD 风格的）“l”参数<sup>1</sup>，可以显示拥有多个线程的进程中的各个线程，Android 中的 ps 工具，只有一个能列出（指定进程中）所有线程的“-t”参数。你可以使用 Android 中的 top(l) 工具，列出各个进程中的线程个数，如输出结果 7-6 所示。

我们可以把上个实验里的那个编程思路套用在这里——你既可以去遍历所有的进程，也可以遍历给定线程组中的所有线程。用 cd 命令切换到 /proc/tgid/task 目录后，你就可以看到一些以数字作为目录名的子目录——它们分别表示（包括主线程在内的）线程组中的各个线程。如果你再用 cd 命令切换到 task/的各个子目录里去的话，你会发现这些目录中的各个文件中记录的信

---

1 原文如此，显然这里应该是“-L”参数，“l”参数表示的是：以长格式输出。——译者注

息和主线程目录中各个文件中记录的信息是十分相似的（实际上/proc/tgid/task/tgid 和/proc/tgid 根本就是一回事）。各个进程和各个线程的目录结构本质上是一样的（再说一遍，在Linux眼里只有线程没有进程），属于同一个进程的各个线程中，几乎所有的进程级属性（比如 maps、fd 等）都是相同的，但是其他一些文件的内容（比如 syscall、wchan 及其他一些文件）却可能是每个线程各不一样的。

```

shell@flounder:/$ top
User 0%, System 1%, IOW 0%, IRQ 0%
User 3 + Nice 0 + Sys 8 + Idle 609 + IOW 0 + IRQ 0 + SIRQ 0 = 620

  PID PR CPU% S  #THR  VSS  RSS PCY U      Name
19213 0  1% R    1 4088K 1340      top
    1 1  0% S    1 1004K  544      /init
..
 154 1  0% S   14 82016K 37568K fg system /system/bin/surfaceflinger
 198 0  0% S    5 4764K  320K   shell
 202 0  0% S    9 12872K 1852K   root
 203 0  0% S    1 1732K  732K   root
 204 1  0% S    1 3556K 1532K   root
 205 0  0% S    2 13932K 5460K fg  drm
 206 0  0% S   11 113916K 24788K fg media
 211 0  0% S    6 2080384K 81204K   root
 212 1  0% S    6 1488620K 59788K   root
 511 1  0% S   82 2258636K 150116K fg system system_server
 691 0  0% S   25 2166460K 141436K fg u0_a20 com.android.systemui
19129 1  0% S   29 2128380K 57220K bg u0_a16 com.android.vending
..

```

输出结果 7-6 使用 toolbox top 命令显示各个进程中线程的数量

status 文件也很容易被搞混，因为这个文件中记录的大多数是针对线程组的信息（所以在属于同一进程的线程中，它们是一模一样的），但其他的一些信息却可能是每个线程都不一样的。比如说，输出结果 7-7 就可以证明 tgid 字段才是真正的进程 ID。

```

shell@flounder:/proc/211/task $ for t in *; do
> echo -n "PID $t: "; grep Tgid: $t/status:
> done
PID 19130: Tgid:      211
PID 19131: Tgid:      211
PID 19132: Tgid:      211
PID 19133: Tgid:      211
PID 19134: Tgid:      211
PID 211: Tgid: 211

```

输出结果 7-7 亲自动手比较 TGID 和 PID

感谢 Android 真正把“给每个线程命名”这一原则落到了实处，你可以把多线程进程中的每个线程的名字都一一列出来，并以此大致推断各个线程的作用。在分析基于 Dalvik 虚拟机的 App（比如输出结果 6-4 中的系统服务），甚至于 Zygote 本身时，这一招还是很管用的，如输出结果 7-8 所示。



```

shell@flounder:/proc/211/task $ for t in *; do
> echo -n "PID St: "; grep Name: St/status;
> done
PID 19130: Name:      ReferenceQueueD
PID 19131: Name:      FinalizerDaemon
PID 19132: Name:      FinalizerWatchd
PID 19133: Name:      HeapTrimmerDaem
PID 19134: Name:      GCDaemon
PID 211: Name:   main

```


输出结果 7-8 显示各个线程的名字

有个并不广为人知的事实：你可以用 `cd` 命令直接切换到指定线程的目录中去，尽管在 `/proc` 目录时，只会显示出各个主线程（和内核线程），但是你还是可以用 `cd` 再加上一个有效的 TID 的方式，直接切换到该线程的目录中去。这样做得到的结果和切换到 `/proc/tgid/task/tid` 里去得到的结果是完全一样的。因为当你 `/proc` 执行 `ls` 操作时，`procfs` 这个伪文件系统会吹毛求疵地把所有子线程（的目录）全都过滤掉。但是当你用 `cd` 命令要求直接切换到某个目录中去时，`procfs` 可就没这么细心了，只要你指定了一个有效的线程 ID，管它是子线程、主线程还是内核线程，你总能切换到那里去。

## 线程状态和上下文切换

线程总是希望，在理想情况下，会一直被执行下去——尽管通常并不需要这样做。线程认为它们的生命周期会在不停的执行中度过，但其实在大部分时间里，它们却只是在等待——等待某个事件发生，等待用户输入，等待 I/O，等待一个互斥锁解锁，甚至只是在等着轮到自己执行 [因为当前所有的 CPU 或者核（core）已经被其他线程占用了]。在任何一个给定的时间点上，内核都维护着一张线程列表，其中记录了每个线程的状态（state）。

---

 进程本身是没有状态属性的。因此，你在每个进程的 `/proc` 目录中看到的状态和上下文切换实时信息反映的实际上都是主线程的这些信息。

---

**state 字段：**`/proc/status` 文件<sup>1</sup>中的这个字段里记录的状态信息，就是 Android 的 `ps` 工具 [或者 Linux 的 `ps` 命令（BSD 风格）] 显示出来的状态信息。这些状态信息的含义和其他 UNIX 系统（比如 Darwin 或其他 BSD 系统）中的十分类似，而且事实上和其他操作系统里的也差不多——甚至包括 Windows 操作系统，尽管相关术语明显是不一样的。图 7-1 给出的这张状态关系图中描述的是各个状态之间的转换关系。

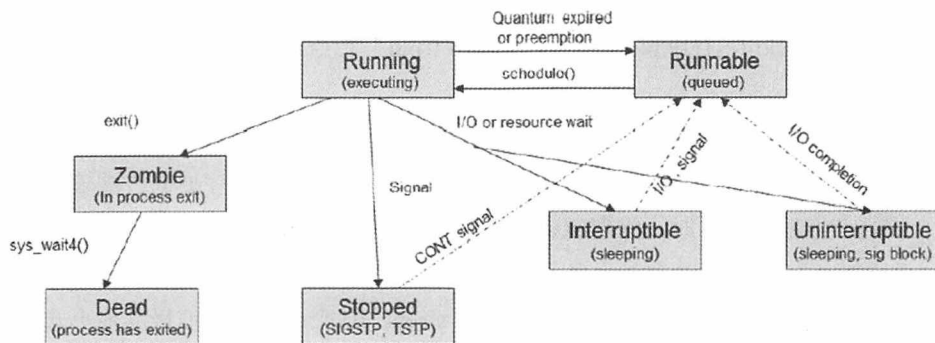
如图 7-1 所示，Running 状态 [即线程当前正在被某个核（core）或超线程核执行] 和 Runnable 状态（即线程当前正处于等待队列中，等待某个 CPU 空闲下来，去执行它）之间是没有明显的

---

1 原文如此，显然是 `/proc/pid/status` 文件之误。——译者注



区别的——从内核的角度看，这两个就是同一个状态。线程会尽可能一直地运行下去，直到下面两种情形至少有一个发生。



内核中的状态的宏名及对应的常量	ps 中显示的值	含 义
TASK_RUNNING(0)	R	进程处于正在 running 或 runnable 状态
TASK_INTERRUPTIBLE(1)	S	进程处于 sleeping 状态，可以（通过信号量）唤醒
TASK_UNINTERRUPTIBLE(2)	D	进程处于 sleeping 状态，不可唤醒
TASK_STOPPED(4)	T	进程（因为收到 SIG_STOP 或 TSTP 信号）进入 stopped 状态
TASK_TRACED(8)	---	进程可以被（单步）调试
TASK_ZOMBIE(16)	Z	进程已经退出，等待资源回收
TASK_DEAD(32)	---	进程已经退出，即将被销毁

图 7-1 Linux 中线程状态的转换机制

- **优先权。**当这种情形发生时，内核会在一个外部中断（external interrupt）的提醒下发现：或者分配给当前执行的线程的时间片已经到点了，或者有一个优先级更高的线程已经变成了 Runnable 状态，进入队列里等待执行了。这时，原来正在被执行的这个线程无疑还想继续运行下去，但系统会把它踢下去，腾出资源好运行下一个线程。这就是大家都一定听说过的“上下文切换”（context switch）。由于这显然是违背了线程自身意愿的，所以也被称为“非自愿式”（nonvoluntary）。
- **休眠/等待。**如果线程在当前实在是无事可做，这一情形就会发生。造成这一情况的原因有好几种，它们分别是：
  - **用户终止了该线程**——终止线程用的是 STOP 信号。UNIX 用户很熟悉的 CTRL-Z 组

合键，实际上就是让终端驱动给主线程发送一个 STOP 信号。这样做就会终止整个线程组（或者说得更顺口些，称之为“进程”）。该进程中的所有线程都会进入 STOP 状态，只有（通常是由 fg 或 bg 发送的）CONT 信号，才能将其唤醒。当然，你也可以亲手输入“kill-STOP pid”或“kill-CONT pid”命令，来终止或唤醒指定进程中的所有线程。

- 终端驱动终止了该线程——因为要在后台运行一个 vi 或者 more 之类的全屏命令，或者在用 stty 启用了 tostop 标志位的情况下，后台中的某个程序要向终端输出信息或从终端获取输入。这时，终端驱动就会向它发送一个 TSTP 信号，（系统的）其他行为和发送 STOP 信号时都差不多。
- 线程主动让出 CPU——如果线程调用了 sleep(2)这个系统调用或以其他形式延迟自己的执行，或者更常见的情况是当线程要等待一个 IPC 对象（即 mutex）时，就会发生这一情形。如果线程要求进行一个不能立即提供服务的 I/O 操作（即，要 I/O 的对象既不在 buffer cache 里，也不在 page cache 里）时，这一情形无疑也会发生。因为 I/O 操作需要和存储设备打交道，或者等待用户本人输入信息，相对于 CPU，这些操作都是极慢的龟速操作。所以，与 I/O 相关的系统调用从大局着眼，会把线程挂起，并把它放入一个等待 I/O 完成的队列中<sup>1</sup>。在 I/O 操作完成之后，（通过一个中断）线程重新进入调度序列，甚至可能因为优先级比其他线程高，而立即得到执行。无论在上述的哪种情形下，线程都是“同意”（或者只是“默许”）进行上下文切换的，这也就是这些情况被统称为“自愿式”上下文切换的原因。

自愿式上下文切换和非自愿式上下文切换之间的区别是很重要的，这也是系统为什么要在 status 文件里提供这一信息的原因。如果一个线程被执行非自愿式上下文切换（也就是在线程还需要执行时，被 CPU “踢了出来”）的次数异乎寻常的多，可能就是在暗示：你需要调高该线程的优先级了（或者只是说明这个线程需要占用 CPU 的时间太长了）。

图 7-1 中给出的最后两种状态 Zombie 和 Dead 并不是真正的状态。这两个状态存在的目的只有一个，那就是进程的返回码。进程的返回码就是使用 exit(2)系统调用时输入的参数，或者 main()函数的返回值。不过这个返回码必须由父进程获取并做相应的处理。这就需要一个负责的父进程（通常是在收到子进程的 SIGCHLD 死亡通知后）以调用 wait#(2)系统调用族中的某个函数的形式，收集子进程的返回码。当进程退出（exit）之后（或者 main(2)函数返回之后），

---

<sup>1</sup> 线程到底是进入 TASK\_INTERRUPTIBLE 状态（在这一状态下，线程仍能接收信号量）还是 TASK\_UNINTERRUPTIBLE 状态（在这一状态下，线程会把发来的信号量挂起），是由系统调用决定的，或者更准确地说是交给负责具体完成这个系统调用的驱动决定的。——原注

主线程就进入了 **Zombie** 状态，等到它的父进程终结它的一切。但如果父进程没能履行其应尽的义务（比如，忽略掉了相关的信号量或者在 `fork()` 之后忘了调用 `wait(2)`——就像某些程序员常常干的那样）子进程就只能进入 **Dead** 状态了。万幸的是，在 **UNIX** 中，僵尸进程（也就是进入 **Zombie** 状态的进程）还是很够意思的，除了要在进程表中占据一条记录之外，它不会占用任何实际的资源——无论是内存、CPU，还是其他的什么资源。当它那个不负责任的父进程也死掉（或者被杀死）之后，僵尸进程或许能被发现，并交给 `init()`（就是 `PID=1` 的那个进程）“收尸”，而 `init()` 总是会乐意调用 `wait#(2)` 系统调用，妥善处理善后的一切。

### 进程级的内存实时使用信息

`/proc/pid/status` 伪文件中还提供了进程使用内存的实时状态信息，这些从整个进程的宏观视角提供的信息是非常有用的（请注意，在这里我们又开始使用“进程”而不是“线程”这个概念了。因为资源分配的主体是进程，而不是线程）。所有这些实时状态信息已经全部列在表 7-1 中了。

表 7-1 `/proc/pid/status` 中记录的高层内存实时使用信息

记录项的名称	含 义
<code>VmPeak</code>	虚拟内存使用量的峰值：就是在该进程的生命周期里， <code>VmSize</code> 取到过的最大的值
<code>VmSize</code>	进程当前已用的虚拟内存的大小
<code>VmLck</code>	被 <code>API mlock(2)</code> 锁定的内存的大小，对于大多数应用程序来说，这一项应该是 0
<code>VmPin</code>	分页锁定内存（Pinned memory）的大小，对于大多数应用程序来说，这一项应该是 0
<code>VmHWM</code>	进程常驻内存集的峰值：也就是在该进程的生命周期里， <code>VmRSS</code> 取到过的最大值
<code>VmRSS</code>	进程当前常驻内存集的大小（也就是对应物理内存页的虚拟内存页大小的总和）
<code>VmData</code>	数据段的大小——这是进程中分配给堆（heap）的内存大小
<code>VmStk</code>	进程分配给各个线程栈的内存大小
<code>VmExe</code>	可执行文件占用的内存大小
<code>VmLib</code>	共享库（.so）文件占用的内存大小
<code>VmPTE</code>	只记录在页表记录项（Page Table Entry）中，但未实际分配内存空间的所有页加起来的大小
<code>VmSwap</code>	进程中被页交换出去的内存大小（在 <b>Android</b> 中，因为没有 <code>swap</code> 也就是页交换机制，所以这一项总是为 0——除非使用了 <b>ZRAM</b> 这类 <code>swap</code> 机制）

查看这些高层信息使用实时信息（特别是高层的 `VmRSS`），可以对内存泄漏（memory hogging）之类的问题，做一个快速诊断。不过为了更深入地分析内存使用中出现的问



还要去看提供了更细粒度信息的`/proc/smaps`<sup>1</sup>，并需要对内存管理的一般性知识有个大致的了解。

## 7.2 用户模式内存管理

程序员一般都不怎么关注内存。因为每个进程都有自己的地址空间，在这个地址空间里，程序可以自由地根据需求分配内存，并且假定内核会把所有细枝末节的问题全部搞定。这个地址空间是私有的（也就是说，它只属于这个进程）和虚拟的（即，它是由内核和内存管理单元从物理内存中抽象出来的）。

不过事实上，内存是应用程序需要面对的最重要的瓶颈之一。内存管理不当不光会对进程本身造成不良影响，还会严重拖累整个系统。在 Android 中，这一问题尤为突出，因为 Linux 的内存管理机制能够在物理内存耗尽时，通过页交换（swap）把一些内存页交换到硬盘上去（尽管页交换机制可能会导致过于频繁的页交换和性能下降，但还是能搞定物理内存不足这一问题的）。但是 Android 里是没有 swap 这个概念的，所以在内存资源耗尽时，必然会触发一个全局性的内存不足（out-of-memory）异常。这时就只能拆东墙补西墙，靠杀掉某个倒霉的进程，释放出它所占用的内存空间，来补救内存不足的问题了。

为了弥补因缺乏 swap 机制而带来的问题，Android 系统在优化利用可用的内存上下足了功夫。Dalvik 虚拟机的设计就非常注重共享尽可能多的虚拟内存。事实也确实如此，如果系统中有多个传统的 Java 虚拟机（比如 Sun 公司的 J2ME）的实例，每个实例至少都需要 100MB 以上的内存。但是如果换成 Dalvik 虚拟机，由于几乎所有的东西都是可共享的，结果每个 App 所需的内存就相当的少。

### 虚拟内存的分类和生命周期

尽管把“所有的虚拟内存都一视同仁”的看法相当的诱人，但事实却并非如此。根据不同的用途以及不同的释放方式，虚拟内存至少可以分成四种类型。内存页自然有它自己的生命周期，如图 7-2 所示。

#### named(mmapped)<sup>2</sup>页和匿名页

内存页的第一种分类方式是根据其中数据的来源进行分类的。从存储设备（磁盘、闪存或

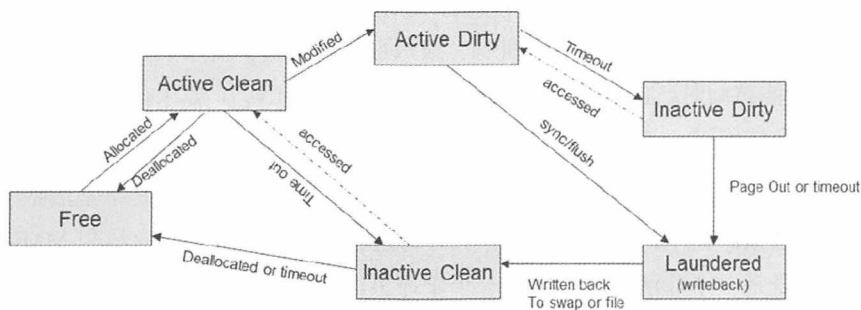
---

1 原文如此，但是`/proc/smaps`这个文件是不存在的！从下文作者一贯的错误来看，此处应为`/proc/pid/smaps`。  
——译者注

2 原文如此，显然括号里是“mapped”之误。——译者注

者网络文件系统)上的文件中得到数据的被称为 mapped 页, 这些页(通过内核的页缓存)从文件中加载数据, 在进程的虚拟内存中需占用一段连续的空间。被 map 的文件的文件名就会成为这些 mapped 页的名字——这也就是为什么这些 mapped 页通常也被称为 named 页的原因。

与 named 页相反, 其他的一些内存页背后是没有对应文件的, 它们是进程在需要时临时创建的。这些页里包括那些用作栈(stack)和用作堆(heap)的内存页, 它们是程序在建立栈帧或调用 malloc(3)时创建的。因为这些页没有对应的文件, 所以也就没有名字——这也就是为什么这些页通常被称为匿名页的原因。



状 态	说 明
Free	该内存页尚未分配
Active Clean	该页最近被一个或多个进程使用过, 但其中存储的内容没有被修改过
Active Dirty	该页最近被一个或多个进程使用过, 且其中存储的内容已经被修改过了
Inactive Dirty	该页最近没有被访问过, 已经“过期”, 且其中的内容已被修改, 应该写回原文件或交换到 swap 中去
Laundered	将该页中的数据写回原文件或交换到 swap 中去
Inactive Clean	该页中的数据已经被写回原文件或交换到 swap 中去了(或者从来没被修改过), 可以重新使用或者释放了

图 7-2 内存页的生命周期

每个进程在 /proc 伪文件系统中的对应目录里都有一个 maps 文件, 其中记录了所有虚拟内存的 map 情况, 因为设备代码和 inode 号以及文件的完整路径都会被明白无误地列在其中, 所以 named 页可以很方便地被认出来。而那些匿名页的这些项上则会被填零或者为空, 当然一些“特殊的”匿名页除外, 比如说被用作栈和堆的页也会被明确地标注出来。

## 脏页和干净页

当一个页被加载到某个进程的内存地址空间中之后, 其中的数据可能会一直没被修改过:

比如被用作只读内存的一部分。进程当然也有可能因为种种原因而修改其他一些内存页中的内容。没有被修改过的页被称为“干净页”(clean)，被修改过的则被称为“脏页”(dirty)。

区分脏页和干净页的意义并不仅仅在于区分数据是否被修改过，在其他一些情况下（比如下面两节中要讨论的，处理活跃页和非活跃页时；或者在决定某些内存页该如何共享时），系统都需要知道该内存页是脏页还是干净页。

## 活跃（active/referenced）页和非活跃页

每个被映射到物理内存上去的虚拟内存页，还附有一个“最近活跃计时器”(age)。内核和 MMU 共同维护了一个引用(reference)机制，跟踪那些最近最少使用的页(LRU, Least Recent Used)。当一个页被访问时，它立刻会被标记成“活跃页”(active)，如果在给定的一段时间里，这个页没有被再次访问过，该页就会被标记成“非活跃页”(inactive)。

在释放内存页(purging)和回写内存页(writeback)操作时，该页是不是活跃的是很重要的。释放内存页是指：当不再需要某个内存页时，进程将该内存页还给内核和 MMU 的操作。如果这个内存页既是一个非活跃页，又是一个干净页的话，那就说明：这个页上的数据不是已经被备份到存储设备上去了，就是全零（也就是没写过）。不管是哪种情况，该页都可以直接被回收掉，然后再次分配给其他需要内存页的进程。

回写内存页是指：进程发现了一个脏页后，把其中的数据写回到存储设备上去的操作。通常，回写操作只有在这个页是映射自某个文件时才能进行，但在一些特殊的情况下，匿名页也会被回写到 swap 里去。在合适的时候，回写 mapped 页是极为重要的，因为系统可能会在毫无预兆的情况下掉电（或者崩溃掉）。没有及时回写 mapped 页会导致数据丢失。所以，内核在 /proc/sys/vm 目录中维护着多个内存页过期(page expiration)参数。匿名页会被写到 swap(ZRAM, 有些 Android 系统支持这种 swap) 里去。如果系统不支持 swap（或称 ZRAM），没有地方可供回写，那么系统就将面临内存不足的问题（这一点将在下文中予以讨论）。

## 私有页和共享页

那些只属于一个进程的内存被称为私有的(private)，也就是说，只有这个进程才能访问这些内存页，没有其他进程能在物理上访问这些内存页。通常，当一个进程用 MAP\_PRIVATE 标志位调用 mmap(2)系统调用 map 某个文件时，或者（在更常见的情况下）用 malloc(3)或 new 分配匿名页时，就会产生私有页。

内存也可以是共享的(shared)，即在两个或两个以上的进程间共享。这一般都是程序员有意执行共享操作而产生的。比如，用 MAP\_SHARED 标志位调用 mmap(2)，或者用其他的机制（比如使用系统中的 shm\* API 或者 Android 中的/dev/ashmem）把内存共享出来。这也被称为显



式共享（explicit sharing），因为进程是为此专门告诉内核“我想和其他进程共享这一段内存”的。

在进程 A 和进程 B 中，同一个共享内存页可能会被映射（map）到不同的虚拟地址上，但它们共享的仍然是同一个物理内存页。这也就是说，共享内存页在系统中只有一份物理记录。毕竟，为什么要再浪费一个物理内存页去存储与另一个内存页里的内容完全一模一样的东西呢？这也使内核在共享内存页里的数据被修改时，不必再去更新其他备份内存页里的数据，因为被映射到各个进程里去的就是同一个物理内存页，所以对它所做的任何修改都会立即反映到（共享该内存页的）所有进程中。

现实情况比这还要复杂那么一点点：有时，进程明明申请的是一段私有内存，但系统却可能会决定要共享这段内存，还不通知发起申请的进程。这就是所谓的“隐含共享”（implicit sharing），隐含共享的例子比比皆是。事实上，除非有特别的说明，大多数的内存都是隐含共享的。比如说，我们来看看库和框架：每个进程（在使用它们之前）无疑都需要先把它们加载到内存里去，而且绝大多数的进程使用的库和框架都是完全一样的，并且不会修改它们。例如，库和框架的代码都是以只读的方法加载的，所以根据相关的定义，它们是不可修改的。要是只要有一个进程需要加载它们（特别是加载 Bionic 之类的通用库或框架）时，就要先复制一份，再把复制出来的页加载到进程里去（因而所有的这些内存页中记录的东西都是完全一样的），那这样做就毫无意义了。即使是在某些特定的情况下，哪怕是在进程明确使用了 MAP\_PRIVATE 标志位的情况下，内核也基本上只是回答一声“好吧，好吧”，然后继续口是心非地以共享的形式加载相关库/框架——明明白白地对进程扯了个谎，赤裸裸地无视 MAP\_PRIVATE 标志位。

为了维持这个精心策划的谎言，内核确实还需要做好一些预防措施：如果有个进程出于某种原因需要修改某个隐含共享的内存页中的内容，内核会允许该进程修改这个内存页，并使这一修改操作不会影响到其他的进程中该隐含共享页中的数据。这时内核会使用一种被称为“写时复制”（copy-on-write）的技术。具体地说就是，内核会[通过一个页错误（page fault）]先中断写操作，然后把这个内存页复制一份，并让复制出来的这个内存页可写，然后把新复制出来的内存页加载到发起写请求的进程中，替换掉原来的那个隐含共享页，而其他进程的地址空间中的内容则保持不变，加载的仍然是原来那个旧的隐含共享页。

### 实验：查看/proc/pid/maps 中提供的内存地址布局

/proc 伪文件系统中，每个进程的目录中都有一个 maps 文件，其中完整地记录了进程内存地址空间的布局情况。它使你能很快地确定哪些文件已经被进程映射（map）了，以及哪些内存段是匿名的。输出结果 7-9 中给出的就是一个已经添加了注释的、（64 位系统中的）shell 进程的 maps 文件中的内容。

```

shell@flounder:/ $ cat /proc/$$/maps
# Most executables are mapped in three segments:
#
#           Write
#           Read|Exec
# Address Range ||| Off
5579579000-55795bc000 r-xp 00000000 103:0d 495 # Code (text), read-only
55795cb000-55795cd000 r--p 00042000 103:0d 495 # Data, read-only
55795cd000-55795ce000 rw-p 00044000 103:0d 495 # Data, read-write
55795ce000-55795cf000 rw-p 00000000 00:00 0 # Anonymous: Reservation
559e8f6000-559e90c000 rw-p 00000000 00:00 0 # Anonymous: Heap
..
7f8298b000-7f82a12000 r-xp 00000000 103:0d 1275 /system/lib64/libc.so # Bionic code
7f82a12000-7f82a22000 ---p 00000000 00:00 0 # Guard (traps overflows)
7f82a22000-7f82a26000 r--p 00087000 103:0d 1275 /system/lib64/libc.so # Bionic constants
7f82a26000-7f82a29000 rw-p 0008b000 103:0d 1275 /system/lib64/libc.so # Bionic data
..
7ff66d9000-7ff66fa000 rw-p 00000000 00:00 0 [stack]
..
Mapping type (p = MAP_PRIVATE, s = MAP_SHARED)

```

输出结果 7-9 通过/proc/pid/maps 查看指定进程（本例中是 shell 进程）的内存地址空间布局

如果还想看更详细的信息，你还可以去看该进程对应目录中的 smaps 文件。该文件中除提供了 maps 文件中的所有信息之外，还有各个内存段的详细属性信息——对于这一点，在下个实验中将会予以简要地演示。

如果要看系统级的内存实时使用情况，你可以去查阅/proc/meminfo 文件，如输出结果 7-10 所示。该文件（top 和 vm\_stat 之类的工具也要使用这个文件）中所使用的术语和分类方法，与刚才我们介绍的是一致的。

```

shell@flounder:/ $ cat /proc/meminfo
MemTotal:          # Total physical RAM in system (2GB, minus reservations)
MemFree: Buffers: 177452 kB # RAM still free
Cached:           336 kB # Buffer cache (used to cache raw device blocks)
SwapCached:       488156 kB # Page cache (used to cache filesystem based I/O)
Active: Inactive: 52 kB # Anonymous memory also in swap
Active(anon):      570104 kB # Total active (broken further to file/anon below)
                 338552 kB # Total inactive - broken further to file/anon below --
                 303228 kB # Total active anonymous memory
Inactive(anon):    130684 kB # Total inactive anonymous memory
Active(file):      266876 kB # Total active mmap(2)ed memory
Inactive(file):    207868 kB # Total inactive mmap(2)ped memory
Unevictable:       1836 kB # Memory locked and unremovable/swappable
Mlocked:           0 kB # Memory locked by processes using mlock(2)
SwapTotal:         520908 kB # Swap: This is a 64-bit L, swapping to compressed RAM
SwapFree:          496780 kB # Available swap
Dirty:            324 kB # Total dirty pages
Writeback:         0 kB # Total scheduled to be written back to files/swap
AnonPages:         422024 kB # Total presently anonymous memory
Mapped:           227664 kB # Total presently memory mapped
Shmem:            11912 kB # Total explicitly shared memory in the system
# ... The rest of the file shows kernel-related memory statistics

```

输出结果 7-10 通过/proc/meminfo 文件，查看系统级的内存实时使用情况

## 内存的相关术语

在计算内存实时使用情况时，你必须把若干因素一并考虑在内，比如：内存是否常驻



(resident)，是否共享(shared)以及其他一些因素。我们先从下面这个简单的公式说起：

$$\text{VmSize} = \text{VmRSS} + \text{VmFileMapped} + \text{VmSwap} + \text{VmLazy}$$

这个公式说的意思是：进程中的虚拟内存可以被划分成互不相干的四大类。

- **VmRSS**: RSS 是常驻内存集大小 (the Resident Set Size) 的缩写。常驻内存页是指那些有对应物理内存页的虚拟内存页。这些虚拟内存页之所以能够进入常驻内存集，可能是因为它们最近是活跃的 (active)，或者在有些情况下，也可能是因为进程 (或者内核) 需要把某些虚拟内存页被锁定 (locked) 在物理内存中 (使这些页不会被交换到 swap 里去)。常驻内存页还可以进一步被细分为：私有的 (Unique，某个进程私有的) 和共享的 (Shared，在两个或两个以上的进程间共享的)。
- **VmFileMapped**: 就是那些其中存储的内容是通过 mmap(2) 系统调用，从文件中提取出来的，在释放内存时可以被写回到文件里去的内存页。事实上，在大多数情况下，如果相关内存页还是干净的 (也就是其中的内容没有被修改过)，在释放内存时只要直接把这些页丢弃掉就可以了。如果今后还需要继续使用这些页中的内容，只要把相关数据再从文件里读出来就可以了，反正它们还都在闪存/磁盘上。这一类内存页所占的空间，并不会直接反映在 /proc/pid/status 文件中，但是我们可以用 /proc/smaps 文件<sup>1</sup> 中的信息把它推算出来——如果要了解的是系统级的相关信息，则可以去查看 /proc/meminfo 文件。
- **VmSwap**: 由于内存分配操作 (即使用 malloc(3) 及类似的函数) 而产生的内存页是没有对应的文件的，所以这些内存页也被称为匿名页——因为它们没有对应的名字 (这个“名字”读成“文件名”更合适些)。所以在必要时，是没有办法把它们备份到文件里去的，这时 swap 分区就跳出来充当“救世主”了——swap 分区是存储器中一个专门用来备份匿名页的分区。不过在 Android 系统中，是没有 swap 分区的，所以这个值几乎总是为 0，除非系统支持使用 ZRAM (经过压缩的内存) 来充当 swap 分区。
- **VmLazy**: 程序员往往都是很贪心的，他们要求分配的内存数量往往比实际需要的多很多。所以内核也针锋相对地使用一种“慵懒” (Lazy) 的方式分配内存，即暂且只在“纸面上”分配内存，直到相关内存页要被使用或者向其中写入数据时，才真正分配物理内存页给它。这些只在“纸面上”得到分配的页会被记录在进程的页表记录项 (Page Table Entry，在 /proc/pid/status 文件中称之为 VmPTE) 中，不过真正的内存分配则会被拖延下来，直到要通过指向该页上某个地址的指针操作相关数据时。这时，内核会收到一个缺页错，它是由 MMU 抛出的——因为这时，MMU 显然会正确地发现，这个内存页不存

<sup>1</sup> /proc/smaps 这个文件是不存在的，显然是 /proc/pid/smaps 之误。——译者注



在。接下来，内核才会真正执行内存页分配操作。使用这个慵懒的分配算法，可以节省大量的内存，但也会对性能造成轻微的影响。最坏的情况是：出现了可用的物理内存页已经没有了，而且也没有哪个页可以写回到磁盘上去（然后予以释放）的情况，在这种情况下，缺页错这个问题就没办法解决。于是处理内存不足（OOM，Out-of-Memory）的代码就会被触发执行——我们将在本章稍后部分讨论这一问题。

大多数人很容易误以为把所有的 VmRSS 加起来，得到的就是被消耗掉的物理内存的大小。可惜的是，事实并非如此，因为进程中有一部分常驻内存页可能是与其他进程共享的。如果只是简单地把所有的 VmRSS 加起来，显然这些页就会被重复计算。所以我们需要一个更加准确的计算方法，这就是 Linux 给出的 PSS（Proportional Set Size，实际使用的物理内存集的大小）实时信息。PSS 的定义用数学语言表述起来是这个样子的：

$$Q_{PSS} = Q_{USS} + \sum_{i=1}^s \frac{Q_{SH_i}}{Q_{P_i}}$$

其中：

$Q_{PSS}$  为实际使用的物理内存集的大小；

$Q_{USS}$  为进程私有的内存页数量；

$s$  为被共享的内存区域的数量；

$Q_{SH_i}$  为第  $i$  个共享内存区域的大小；

$Q_{P_i}$  为共享了第  $i$  个共享内存区域的进程的数量。

如果你像大多数非数学专业人士那样，有那么点公式恐惧症（或者你发过誓，再也不碰“西格玛”之类的符号了），那么这个公式也可以用下面这样的话来表述：

- 对于进程私有的内存页（USS）来说，有 1KB 算 1KB，全部累加起来算入 PSS——也就是说，如果一个内存页是某个进程私有的，它就应该被不打折扣地算入 PSS 中。
- 对于一个被  $n$  个进程共享的内存页来说，每 1KB 内存，在 PSS 中只能算  $1/n$  KB，其中这个  $n$  是共享这个内存页的进程的数量。因为进程中可能有多个共享内存区域（或段），所以我们需要使用“西格玛”符号——也就是说，在 PSS 中加上第一个共享内存区域大小的  $1/n_1$ ，再加上第二个共享内存区域大小的  $1/n_2$ （ $n_2$  为共享了第二个内存区域的进程的数量），依此类推……

乍一看，PSS 中竟然有分数？这可真是够奇怪的！但是如果你把系统中所有进程的 PSS 加在一块儿，神奇的数学就会把一切都搞定，每个共享内存段都会被记入 PSS 且只记入一次，不

多也不少，精确地算出系统物理内存的使用量[如果你想来点小小的刺激，你可以手工累加一下(两个)“西格玛”符号的计算结果，证明这个结果的正确性]。

幸运的是，估算 PSS 是非常有用的，为了得到它，即使再复习一下代数公式也是值得的，更何况，PSS 还可以直接从`/proc/smmaps1`文件字节中读取出来(完全不需要任何数学知识)。下面这个实验就是最好的例子。

### 实验：通过`/proc/pid/smmaps`文件查看 RSS、USS 和 PSS

每个进程在`/proc`伪文件系统中的 `smaps` 文件和 `maps` 文件一样，都是以内存段的形式显示内存的实时使用情况的，它们的区别在于：`smaps` 文件中还提供了每个内存段更详细的信息。在这个实验中，我们要找这样一个实验用的样本程序：这个程序当前在系统中应该没有另一个进程实例；而且这个程序执行起来还应该需要一段时间，不会一闪而退——这是为了方便我们操作。一个很好的选择就是 `ping`，它是一个真正的二进制可执行文件(没有被集成在 `toolbox` 里)，而且可以一直运行下去。

执行 `ping` 程序时，我们要给它输入一个 IP 地址——我们甚至连这个 IP 是不是可达的都无所谓，只要它能让 `ping` 程序一直运行下去就可以了。如果你选择使用其他二进制可执行文件来做这个实验，也没有关系(具体选用哪个样本程序对实验不会产生任何影响)，只要这个二进制可执行文件能够被系统加载，创建出一个进程来，然后你能够挂起这个进程就行了。一旦程序运行起来之后(它可能会停下来，等待用户输入数据)，按下 `CTRL-Z` 组合键把它挂起来并回到命令行提示符下，或者你也可以把它扔到后台去运行。然后，我们来查看一下 `smaps` 文件的前 10 行左右的数据。如果你在实验中使用的是样本程序是 `ping` 的话，查看的结果就应该和输出结果 7-11(a)差不多。

```
# Start the binary in the background - collect PID
shell@flounder:/system/bin $ ping 1.1.1.1 > /dev/null &
[1] 20117
shell@flounder:/system/bin $ more /proc/20117/smmaps # inspect ping's smaps entry
55705e3000-55705ec000 r-xp 00000000 103:0d 463 /system/bin/ping
Size: 36 kB
Rss: 32 kB
Pss: 32 kB
Shared_Clean: 0 kB
Shared_Dirty: 0 kB
Private_Clean: 32 kB
Private_Dirty: 0 kB
Referenced: 32 kB
Anonymous: 0 kB
AnonHugePages: 0 kB
Swap: 0 kB
KernelPageSize: 4 kB
MMUPageSize: 4 kB
Locked: 0 kB
VmFlags: rd ex mr mw me
dw ..
```

输出结果 7-11(a) 查看指定程序只有一个进程实例时的 USS、RSS 和 PSS

1 原文如此，问题是没有`/proc/smmaps`这个文件的，显然是`/proc/pid/smmaps`之误。——译者注

那么，我们该怎么解读这个输出结果呢？

- 第一个内存段是从存储设备加载到内存里来的/usr/bin/ping 程序（这个就不用大惊小怪了吧）。这个内存段是可读、可执行而且（好像是）进程私有的（r-xp）。它是从设备 103,0d 那里加载上来的，inode 编号为 463。
- 这个内存段的 VmSize 是 36KB，其中有 4KB 应该刚刚被释放掉——因为这个段的 RSS 是 32KB。被释放掉的应该是 ELF 头的一部分——因为这些数据在程序的运行过程中不再会使用了。
- 剩下这 32KB 的 RSS 都是私有页和干净页：私有页，意味着这些内存是这个进程私有的（即 USS）；干净页，意味着内存中的数据从被加载到内存里之后，一直到现在都没有被修改过。所有这些内存页也是最近活跃的（/proc/pid/smmaps 文件中记录在 reference 一栏中），这一点也没什么可奇怪的，因为 ping 程序现在正在执行。这些内存页也都不是匿名页（因为它们都加载自某个文件）。
- 所以 PSS 就是 32KB。因为这个段中没有共享内存页，所以每个 USS 内存页都应该算入 PSS 中。

到目前为止，一切都似乎完美无缺。但是如果我们现在再打开一个 ping 程序（或者你选择的样本程序）的进程实例，又会发生些什么呢？我们来这样做一下，然后再去查看第一个（不是第二个！）进程的 smaps 文件中的内容，我们看到其中的内容确实发生了变化！如输出结果 7-11（b）所示。


```
# Once again, start the binary in the background - collect PID
shell@flounder:/system/bin $ ping 1.1.1.1 > /dev/null &
[2] 20130
shell@flounder:/system/bin $ more /proc/20117/smmaps
55705e3000-55705ec000 r-xp 00000000 103:0d 463      /system/bin/ping
Size:                36 kB # VmSize unchanged
Rss:                  32 kB # RSS unchanged
Pss:                   16 kB # PSS drops by half because
Shared_Clean:         32 kB # All 32k are now shared!
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:         0 kB
Referenced:           32 kB
Anonymous:             0 kB
AnonHugePages:         0 kB
Swap:                  0 kB
KernelPageSize:       4 kB
MMUPageSize:          4 kB
Locked:                0 kB
VmFlags: rd ex mr mw me
dw ..
```

输出结果 7-11(b) 查看指定程序有两个进程实例时的 USS、RSS 和 PSS

比较这两个输出结果，你可以看到：第一个进程里大多数的记录并没有发生变化，VmSize 还是 36KB，RSS 还是 32KB，不过因为 RSS 现在全部是共享页了——在两个 ping 程序的进程实例之间共享。所以 PSS 应该减半，变成 16KB。



如果再运行一个 ping 程序的进程实例，会把 PSS 进一步降为 10KB（从技术上说，应该是 10.6KB，但是被取整了），如果系统中有四个该进程的实例，PSS 会再降为 8KB（32KB 可以被 4 整除）。如果杀掉几个进程实例，就等于是减少了共享这个内存段的进程的数量，会导致 PSS 的值回升。

 在这个例子的整个过程中，这个内存段好像是私有的（因为在所有的输出结果中，段的权限项都是 r-xp）。但是这显然是错的，因为 smaps 明确地显示这个内存段是共享的。出现这个悖论的原因是：r-xp 中的 p 表示的并不是 private（私有的），而是 MAP\_PRIVATE——它是调用 mmap(2) 映射（map）这个内存段时的参数。这个问题在之前讲解私有页/共享页时其实已经讲过了。这种情况有个专门的规定：进程可以用 MAP\_PRIVATE 标志位映射一个内存段，而且操作系统确实也会答应这么做，但一旦有另一个进程也映射了相同的内容，内核会保留在两个进程间隐含共享这个内存段的权力，直到两个进程中有一个进程要修改内存段中的内容（把相关内存页变成脏页）为止。如果真的发生了这样一个写操作，执行写操作的进程就会触发一个写时复制错，这会导致内核再分配一份该内存页的拷贝（供修改），防止原有的内存页中的数据被破坏。与此相反的权限标志位的对应字符是“s”（它是 mmap(2) 的参数 MAP\_SHARED 的缩写），也就是显式共享——表示相关内存页只有一份拷贝，其中的内容可以被修改（弄脏），而修改的结果则会在各个进程间共享。

这个实验有望成为演示 PSS 是如何计算的一个简化了的样例。说它是“简化了的”是因为在这个例子中，所有的内存页都是共享的，因此就直接把 USS 降为 0 了，这使得 PSS 的计算相当简单。但是内存中的其他一些部分可就复杂多了，其中既有私有页又有共享页，这就使得计算 PSS 更具一些挑战性。不过万幸的是，smaps 文件里提供了已经自动算好了的 PSS。

如果你不是手工解析 /proc/smaps 文件<sup>1</sup>的狂热爱好者，那么在下面这个实验里，我可以给你介绍一个工具——啊不！事实上是两个工具。

### 实验：使用 procrank 和 librank 查看 RSS、USS 和 PSS

AOSP 提供了两个能查看内存实时使用情况的很有用的工具 procrank 和 librank，不过在大多数手机/平板电脑里，这两个工具都会被厂商删掉。不过，把这两个程序连同它们的依赖库 /system/lib/libpagemap.so 一起从模拟器镜像里拷出来，然后再复制到手机/平板电脑上去，也不是什么难事。整个过程如输出结果 7-12 所示。

1 原文如此，再说一遍，/proc/smaps 是不存在的，显然为 /proc/pid/smaps 之误。——译者注

```
morpheus@forge (~/.tmp)$ adb -s emulator-5554 pull /system/xbin/procrank
morpheus@forge (~/.tmp)$ adb -s emulator-5554 pull /system/xbin/librank
morpheus@forge (~/.tmp)$ adb -s emulator-5554 pull /system/lib/libpagemap.so
# kill emulator or use -s with serial number of device from adb devices
morpheus@forge (~/.tmp)$ adb push librank /data/local/tmp
morpheus@forge (~/.tmp)$ adb push procrank /data/local/tmp
morpheus@forge (~/.tmp)$ adb push libpagemap.so /data/local/tmp
```

输出结果 7-12 把二进制可执行文件从模拟器复制到手机里去

复制完毕之后,你先要在手机上把相关二进制可执行文件的权限设为可执行(用 `chmod(1)`),然后就能运行它们了。不过因为依赖库也是被复制到 `/data/local/tmp` 目录里的,但是库的默认搜索路径是 `/system/lib[64]`,所以你还修改一下库加载路径,如输出结果 7-13 所示。另外,如果你的手机已经 root 了,也可以直接把依赖库 `libpagemap.so` 复制到 `/system/lib` 里,这样就不必修改库加载路径了。

```
# On device:
shell@htc_m8w1:/ $ chmod 755 /data/local/tmp/procrank
shell@htc_m8w1:/ $ /data/local/tmp/procrank
CANNOT LINK EXECUTABLE: could not load library "libpagemap.so" needed by
"/data/local/tmp/procrank" caused by library "libpagemap.so" not found
shell@htc_m8w1:/data $ LD_LIBRARY_PATH=/data/local/tmp /data/local/tmp/procrank
```

PID	Vss	Rss	Pss	Uss	cmdline
234	331600K	89211K	61314K	49124K	system_server ..

输出结果 7-13 把二进制可执行文件设为可执行,并修改库加载路径

在复制完 `procrank` 和 `librank` 之后(或者你也可以直接在模拟器里运行这两个工具),你就可以开始分析它们的输出结果了。这两个工具都会依次去读取 `/proc` 伪文件系统里各个进程对应目录中 `smaps` 文件里的内存实时使用信息(在比较新的版本里,它们还会去读取 `pagemap` 文件中的数据)。但是在如何输出这些内存实时使用信息这个问题上,这两个工具还是不一样的:`procrank` 是根据各个进程所用内存的大小,把各个进程降序排序后输出的;而 `librank` 则是按照各个内存段所占 `RSS` 的大小,把各个内存段降序排序后输出,而且输出每个内存段时会同时输出共享了该内存段的所有进程。从这两个工具的输出结果中,我们可以得到大量关于 Android 系统内存使用(和优化)的信息。我们先来看看 `procrank`,见输出结果 7-14。

如输出结果 7-14 所示,进程确实是按照其所用 `VSS` 的大小降序列出的(这是因为使用了默认的参数 `-v` 所致,你也可以用参数 `-p`、`-r` 或者 `-u`,分别要求按进程所用的 `PSS`、`RSS` 或 `USS` 的大小依次列出各个进程)。其他参数还包括只显示 `cached` 内存页(`-c`)和 `non-cached` 内存页

(-C)<sup>1</sup>，你也可以使用-h 参数列出 procrank 的所有参数。

```
root@generic:/ # procrank
PID      Vss      Rss      Pss      Uss      cmdline
354      631600K  99212K   60712K   48784K   system_server
709      578240K  91200K   45218K   24512K   com.android.systemui
581      565940K  72876K   43295K   38940K   com.android.launcher
52       102852K  47784K   24356K   2132K    /system/bin/surfaceflinger
538      540284K  44108K   16408K   13284K   com.android.phone
66       508532K  46416K   14799K   8948K    zygote
955      531912K  30760K   6756K    4700K    com.android.calendar
843      522168K  30364K   6264K    4396K    com.android.providers.calendar
1077     521044K  28308K   499K     3668K    com.android.browser
978      520144K  29004K   311K     3336K    com.android.deskclock
1037     520732K  27380K   82K      3484K    com.android.exchange
```

输出结果 7-14 在 Android L 模拟器中执行 procrank 命令得到的输出结果

不过，在上面这个输出结果中，有些结果相当的扎眼，比如 launcher 进程的 VSS 的值非常大（565MB！Launcher 真的需要用这么多内存吗？），但是该进程实际占用的 RSS 就小得多了（这表示其中有大量的 named 页已经被丢弃掉了），当然它的 PSS 就更小了。另外，你也可以看到，每个 App 的私有页所占的物理内存空间（USS）平均都不超过几 MB！平均每个 App 里有大约 85%~90% 的 RSS 是共享的，这极大地减小了相关进程的 PSS 的大小。这些共享部分大多是 Zygote 和 Dalvik 虚拟机（包括 ART）的组成部分——这一做法最大化地利用了共享内存的优势，把 Java 虚拟机远远地甩在了身后（但可能有些人还是会跳出来争辩说，这还不如 iOS 有效）。

你可以回过头去，用 RSS 减去 USS 算出共享内存的大小，用 PSS 减去 USS 算出的是共享内存的加权平均大小。然后再用共享内存的大小除以共享内存的加权平均大小，你就能大致算出有多少个进程共享着这些共享内存。之所以说你只能“大致算出”是因为：在把所有的 PSS 加起来的时候，会损失一些精度——因为共享不同内存段的进程的数量不一定是一样的。

Librank 工具的输出结果就略有不同了，因为它是按内存段排序的。不过除了这一点之外，所有术语的含义都是一样的。输出结果 7-15 显示的是加载了 boot.oat（这个二进制可执行文件

1 按 procrank -h 中给出的解释，cached page 就是 storage backed page，non-cached page 就是 ram/swap backed page，分别指已经备份到闪存里的文件中的页，和正在内存中及已被页交换到 swap 中的页。乍一看很奇怪，已被页交换到 swap 中的页怎么会和正在内存中的页搅在一起呢？原因是因为闪存读写次数的限制，不适宜用作 swap，所以 Android 中的 swap 是内存中单独专门划出的一段空间充作 swap 的，被交换到这个 swap 中的内存页中的数据会先被压缩，然后再存放到这个 swap 中，这样这个 swap 就能存储比正常情况更多的数据。由于不论是正在内存中的页，还是被交换到 swap 中的页，它们都是在物理内存中的，所以就被称为 non-cached page。——译者注



中存储的是一些预编译好的 ART 框架类) 的内存段使用情况。

```

root@generic:/ # librank
RSStot  VSS      RSS      PSS      USS  Name/PID
..
27179K
48556K  19576K  6468K  3268K  system_server [354]
48556K  13488K  3982K  2112K  zygote [66]
48556K  13984K  3802K  2192K  com.android.phone [538]
48556K  14800K  3164K  744K   com.android.systemui [709]
48556K  11880K  1933K  324K   com.android.launcher [581]
48556K  10544K  1302K  160K   com.android.inputmethod.latin [500]
48556K  9268K   958K   104K   android.process.media [766]
48556K  9048K   861K   64K    com.android.email [997]
48556K  8200K   826K   96K    com.android.server.telecom [532]
48556K  8628K   759K   20K    com.android.calendar [955]
48556K  7960K   655K   12K    com.android.providers.calendar [843]
48556K  7280K   588K   24K    com.android.deskclock [978]
48556K  7220K   566K   20K    com.android.browser [1077]
48556K  6772K   474K   0K     com.android.exchange [1037]
48556K  6132K   429K   0K     com.android.dialer [1061]
48556K  5796K   405K   4K     com.android.sharedstoragebackup [1096]

```

输出结果 7-15 librank 的输出结果展示了 ART 预编译类的共享情况

librank 再一次显示了共享内存的有效性: boot.oat 这个段的 VSS 大约是 48MB, 但实际上, 其中属于常驻内存集的连一半都不到, 而且对于大多数进程来说, 在这个 oat 段中, 进程私有内存所占的物理内存空间都只有几十 KB 而已。

更新一些版本的 Android 系统中甚至还使用了一种更有效的内存共享方式——Linux 内核中的 KSM (Kernel Samepage Merging, 内核相同页合并) 机制。这一特性使得内核能够 (通过比较 Hash 的方式) 自动检查物理内存页中的数据是否相同——相关物理内存页甚至可以还没被 map 到某个虚拟内存地址上。如果找到了两个物理内存页中的数据是完全相同的, 这两个页就会在引入了“写时复制” (copy-on-write) 的限制后, 被合并。自 2.6.27 版本开始, KSM 已经成为 Linux 内核的一个特性, 但是直到最近它才被引入 Android 系统中。

## 内存不足时的应对方案

尽管 Android 最大限度地利用了共享内存的优势, 并佐以 KSM 和 ZRAM 等技巧, 但没有真正的 swap 空间仍然是一个绕不过去的坎。这并不是 Android 设计上的缺陷, 因为闪存芯片的刷写次数有一定的上限, 所以闪存和 swap 就是没法一起使用。由于没有 swap, 系统在任何时候都可能会出现物理内存耗尽的情况——这是一个相当明确且现实的危险。

Linux 内核很早就有了一个处理内存不足的机制。这个被称为 OOM (Out-Of-Memory, 内存不足) 的机制会在系统无法满足内存分配请求时被触发。不过在 Linux 中, 它被触发的情况很少——如果系统剩余物理内存空间低了的话, 通常都有大量的 swap 空间可供页交换。所以只有当物理内存和 swap 空间都不足时, 才会触发 OOM。

OOM 并不是一个线程, 它被实现为发生内存不足的情况时, 要被执行的一系列代码。这些

代码逐一遍历所有的进程，试图从中找出一个最合适的“受害者”——杀掉这个进程能在最大程度上缓解系统内存不足之苦。所有的进程都作为候选对象，在这个“死亡队列”中，以 `oom_score` 值（这是一个启发式的评分测试分值，具体评分测试的算法一直随着内核版本的更新而不断地调整）排序。这个分值也可以在 `/proc/pid/oom_score` 这个只读的伪文件中读取到。

成也启发式，败也启发式，`oom_score` 的问题在于：它好像并不是总能可靠地工作，常常会有无辜的进程被误杀掉——只是因为它在错误的时间得到了一个错误的 `oom_score` 分值。由于杀掉被选中的进程的操作（实际上就是个 `kill -9`）会被立即而果断地执行（没有任何回转余地），所以“受害者”进程是无法做出任何反抗的。

因此，Android App 的整个生命周期总是笼罩在一层随时随地会被莫名其妙地干掉的阴影中。应用无法以任何一种形式保证自己能够看见明天的太阳，它们只能使用一些回调函数来保存它们的状态（还是通过文档不详细的 `android.os.Bundle`），系统也只能保证：如果它被杀掉了，当它再次执行时，可以使用这个 `bundle` 中的信息恢复之前的状态。应用程序根本无法预测自己何时会被杀掉，甚至连自己会不会被杀掉也都无法预测。而在 iOS 里对应的 `jetsam`（一个设计用来完成类似功能的机制）中，应用甚至连这个 `tombstone`<sup>1</sup> 也留不下来（尽管在内核日志中会保留一些相对详细的信息）。

为了稍稍缓解一下这个因启发式评分方式引发的“宿命论危机”，Linux 提供了一种在用户空间中调整 `oom_score` 的方式，它就是 `/proc/pid/oom_adj` 和（在比较新的内核版本中的）`/proc/pid/oom_score_adj` 文件。这两个文件让用户空间里的进程能够给 `oom_score` 加上一个修正值——如果加上的是个负的修正值，就会把 `oom_score` 变小（因而会降低进程被杀掉的概率）；如果加上的是个正的修正值，就会把 `oom_score` 变大（这对那些有自杀倾向的进程倒是蛮有效的……）。

Android 的系统进程使用这一机制，确保它们不会被杀掉：`/init` 和它根据各个 `.rc` 文件启动的同级进程，会在自己的 `oom_adj` 加上一个 -16 或 -17 这样的修正值（这使得 OOM 完全不会杀掉它们）。在比较新的内核里，它们给 `oom_score_adj` 加上一个 -1000 的修正值，以达到相同的目的——这个值会让它们的 `oom_score` 值接近于 0（如果不是等于 0 的话）。

从坏的角度讲，这一功能可能会被 App 滥用（毕竟，谁也挡不住永生的诱惑不是？），为了防止这一手，Android 的 `ActivityManager` 会在应用生命周期里的不同阶段，自动重置 `oom_adj/oom_score_adj` 中的修正值（这一点我们将在第 2 本中予以讨论）。在 Android L 及以后的版本中，`ActivityManager` 是通过 `lmkd` 服务调整 OOM 分值的（相关讨论见第 5 章），因为

---

1 `tombstone` 意为墓碑，这里指 Android App 进程被杀掉时留下的这个 `bundle`。——译者注

oom\_adj/oom\_score\_adj 文件只为 root 所有，而且也只有 root 才有写它的权限。

Android 提供的另一种预防措施是以 LowMemoryKiller (lmk) 的形式给出的。这是一种抢在真正的 OOM 被触发执行之前，先杀掉一些进程的防止 OOM 误杀“好人”的 Android 技巧。在早期的 Android 版本中，init 会在系统启动时，设置 sysfs<sup>1</sup>中 module/目录下的各个参数。自从 Android L 开始，init 只管确保 sysfs 中各个伪文件的访问权限的设置是正确的，设置 sysfs 中 module/目录下的各个参数的任务就留给 lmkd 了。

### 实验：切换 App 至后台的操作对 oom\_score 修正值的影响

你可以通过查看 /proc 中进程对应目录中相关文件的内容，实时观察 oom\_score 值的变化情况。在这个实验中，我们需要在打开一个 App 的同时，再打开一个 adb shell。例如，如果你选用的样例 App 是 Chrome Web 浏览器，你打开 adb shell 之后，可以做如输出结果 7-16(a)所示的操作。

```
root@flounder:/ # ps | grep chrome
u0_a34      12079 211    2295800 167140 ffffffff a7058b1c S com.android.chrome
root@flounder:/ # cat /proc/12079/oom_score_adj
0
root@flounder:/ # cat /proc/12079/oom_adj
0
root@flounder:/ # cat /proc/12079/oom_score
70
```

输出结果 7-16(a) 观察一个正在前台运行的 App 的 oom\_score

现在，把这个 App 切换到后台去（你只要按一下 Home 键就可以了），这个操作的影响将会自动反映到 OOM 上：oom\_score 的修正值将会变大，因此 oom\_score 的值也会变大（见输出结果 7-16(b)）。

```
root@flounder:/ # cat /proc/12079/oom_adj
6
root@flounder:/ # cat /proc/12079/oom_score_adj
411
root@flounder:/ # cat oom_score
470
```

输出结果 7-16(b) 观察一个被切换到后台运行的 App 的 oom\_score

在 Android L 中，你还可以在 App 的运行过程中，在 lmkd 进程上附加一个调试器 (trace)，观察 ActivityManager 发送给 lmkd 的消息。具体操作方法已经在第 5 章的实验中（特别是输出

<sup>1</sup> sysfs 的具体作用详见第 2 章。——译者注



结果 5-6 中)演示过了。此外,挂起 `lmkd` (用 `kill -STOP` 命令)会使所有这类调整 `oom_score` 修正值的行为都不会生效。

## 7.3 跟踪系统调用

事实上,一个用户模式下的线程要想完成任何“有意义”的操作,都必须要请求系统以某种方式介入。无论是处理文件,打开 `socket` 还是对(除了它之前已经分配到的虚拟内存之外的)资源做任何类型的处理,用户态线程总是需要请求来自内核的服务——也就是系统调用(`system call`)。

使用系统调用首先需要用户模式的进程进入内核模式,具体的进入方式随处理器的体系结构的不同而各不相同,不过这总是会涉及某条特殊的机器指令——在 ARM 处理器中它是 `SVC` 指令(又名 `SWI` 指令);在 Intel 处理器中它是 `SYSENTER` 指令(或者 `SYSCALL` 指令)。这些指令会把处理器的模式改为特权(`privileged` 或 `supervisor`)模式,并把系统控制权转移到系统启动时就定义好了的内核入口点(`system_call`)上。因此,所有的系统调用会先被交到一个函数手上,这个函数再用(通过 ARM 中的 `r12` 寄存器,或者 Intel 中的 `EAX` 寄存器传来的)系统调用号(`system call number`)查询一张内部的(系统调用地址)表,把执行流跳转到提供对应服务的系统调用函数那里。

了解了上述这些知识之后,我们就能明白,为什么在调试(`debug`)或跟踪(`trace`)进程时,系统调用是这么的重要。在大多数情况下,对进程内部的操作(修改这个或那个变量的值)我们其实并不是很关心,或许只是因为这些操作的数量非常大且难以追踪的缘故吧。但是对文件或 `socket` 的操作就特别地关注,而跟踪系统调用就提供了一种在混杂了大量其他操作的情况下,追踪这类操作的简单方法。

### toolbox ps 工具

`toolbox ps` 工具的输出结果中有两列非常有价值(尽管比较简略!)的关于系统调用的信息:`WCHAN` 和 `PC`。前者是“Wait Channel”的缩写,它表示该进程(或内核线程)当前在内核中(执行到)的地址,如果不能确定的话,则记为-1(`0xffffffff`) (回忆一下,除非使用的是 `ps -t` 命令, `ps` 输出结果中的每一行表示一个内核线程或某个进程的主线程)。后者则是(用户态里的)返回地址,也就是系统调用执行完毕返回之后,进程该从哪里开始继续执行。解析 `WCHAN`

---

1 不光简略,还有 bug。直到 Android M,在使用 64 位处理器的手机/平板电脑中, `ps` 并不能正确地显示 `PC` 的值——它还是认为 `PC` 应该是个 32 位的值。——原注

中记录的内核地址是属于哪个内核函数的，需要手工操作一番，如下面这个实验所示。

**实验：手工解析 toolbox ps 命令的执行结果中 WCHAN 一列中记录的地址属于哪个内核函数**

在面对一个 WCHAN 地址或者其他内核地址时，你可以用这里演示的简易方法，用 /proc/kallsyms 文件，解析出该地址所属内核函数的可读性更强的函数名。我们先从某个确定的要查询的地址开始（应该不会这么巧，立刻就能查到与这个地址对应的某一个函数名）。因为 kallsyms 文件中只记录了各个内核函数的起始地址，而 WCHAN 上记录的是当前执行到的地址，一般都是函数内部的某个地址，只有在很小的概率下，才会正好执行到函数的起始地址。这时，你应该回过头去，去掉最后一位数字，用 grep 去搜索所有符合这个前缀的行，如果还是没有匹配的结果，就应该再回去去掉最后一位数字，再用 grep 去搜索，如此往复，直到有匹配的项出现为止。在有匹配结果出现时，可能只出现了一个匹配结果，也有可能出现多个匹配结果，这时，最接近我们搜索的地址的那个结果中记录的就是我们要找的内核函数的函数名（见输出结果 7-17）。

```
# Prerequisite - disable kernel pointer hiding (0 - fully disable, 1 - root only)
root@generic:/# echo 0 > /proc/sys/kernel/kptr_restrict
# Now, attempt to get the address. Note the first few attempts fail
1|root@generic:/data # grep c0029d4 /proc/kallsyms
1|root@generic:/data # grep c0029d /proc/kallsyms
1|root@generic:/data # grep c0029 /proc/kallsyms
c00294f0 T exit_signals
...
# Too many results - go back by 1 hex digit (i.e. d - 1 = c)
root@generic:/data # grep c0029c /proc/kallsyms
c0029c14 T do_sigtimedwait # Got it!
```

输出结果 7-17 用 /proc/kallsyms 文件解析内核地址所属函数的函数名

必须要说明的是：这个“最接近”的结果必须是我们搜索的地址之前的，而不是之后的地址。有时，这个最接近的地址就会和其他许多匹配结果混在一起被输出出来（就像本例中这样）；有时，grep 输出的匹配结果可能全部在你搜索的这个地址之后（因此全都是不正确的）。



Android M 预览版中的 toolbox 会通过查询 /proc/<pid>/wchan 文件（这个文件将在下一节中讨论），自动把 WCHAN 中的地址映射为对应内核函数的函数名。但即使是这样，这个实验也还是有意义的，因为它向你演示了解析内核地址的相关技术。

## wchan 和 syscall 文件

/proc 伪文件系统中也提供了系统调用的跟踪机制。在每个线程对应的目录中都有一个 wchan 文件，其中记录了该线程休眠时在内核中执行到了什么地址（如果该线程当前是活跃的，

其中记录的就是 0)——其输出的内容和 toolbox ps 命令输出的十分类似,只是 wchan 的输出已经自动解析为“最近的”内核函数名了。这使你能够免于不停地重复上面那个实验之苦,而且即使是在 /proc/kallsyms 文件有访问限制时,它也能正常工作。

在某些版本的内核中,每个线程对应的目录中还有一个 syscall 文件,其中提供了更详细的信息:这个文件中记录了该线程在当前这次轮询发生时,所请求的系统调用号及相关参数。syscall 文件记录这些数据的格式,如输出结果 7-18 所示。

```
root@flounder:/proc/12079 # cat syscall
# num   Arg1   Arg2   Arg3   Arg4   Arg5   Stack Pointer   Program Counter
22      0x10 0x7fc139b110 0x10 0x2324a 0x0 0x8 0x7fc139b040 0x7fa7058b1c
# syscall 22 in ARM64 is epoll_wait, as confirmed by wchan:
root@flounder:/proc/12079 # cat wchan
Sys_epoll_wait
```

输出结果 7-18 /proc 伪文件系统中的 syscall 和 wchan 文件

你也可以使用上个实验中给出的方法,解析程序计数器(program\_counter)的值——也就是 toolbox ps 的输出中 PC 一列中记录的数据<sup>1</sup>。不过要说明的是:在针对不同体系结构的处理器编译的系统中,各个系统调用号对应的系统服务不一定是一样的。Intel 处理器和 ARM 处理器使用的系统调用号不一样是完全可以理解的,可令人想不到的是:即使都是 Intel 或 ARM 处理器,32 位处理器和 64 位处理器使用的系统调用号也是不一样的。在 Android NDK 中的 platforms/android-APIversion/arch-arch/usr/include/asm/unistd.h 文件(其中斜体的 arch 应该被替换为 arm、arm64、x86 或 x86\_64)中,可以查到你所用的处理器对应的各个系统调用号的含义。幸运的是,提供 syscall 文件的内核通常也都提供 wchan 文件,所以你可以像上面这个输出结果中那样,根据 wchan 文件中的记录,推出系统调用号的含义。大多数内核还会提供一个 stack 文件,其中记录了内核栈中的详细信息。

## strace 工具

到现在为止介绍的方法都是基于轮询的,也就是说,你确实可以准确地读到某个系统调用正被使用,但是由于每次查询都要发起读取操作,所以你一次只能抓取到一个系统调用。这一方法在诊断已经被挂起或者失去响应的进程时是很有用的。但是,大多数系统调用跟踪最好是不间断地进行,尽可能隐蔽地附加进程,并在每次系统调用被执行时都得到通知。

1 这里作者晕头了?用上个实验的法子去解析 PC 的值?上个实验中使用的 /proc/kallsyms 文件只记录了各个内核函数的函数名和函数起始位置,但是 PC 中的值是在用户态下的!所以 PC 的值只能用符号文件或者逆向工程的方法才能知道它属于哪个函数以及有什么意义!——译者注



这时就该 `strace` 工具闪亮登场了。这是个非常强大的程序，本书中到目前为止已经多次用它跟踪和解释进程内部的行为了，作为一个跟踪（`trace`）工具它确实是个无价之宝。这个工具的完整使用方法本身就足够写上整整一章的内容了，所以笔者只能在表 7-2 中综述了这个工具最有用的几个参数的用法。

表 7-2 `strace` 最有用的几个参数的用法

参 数	用 法
<code>-i</code>	输出系统调用的入口指针
<code>-t [t[t]]</code>	以不同的时间格式和精度，在输出中的每一行前加上时间信息
<code>-f</code>	在目标使用 <code>clone()</code> 系统调用时，自动附加子进程/线程
<code>-o file</code>	把输出结果保存到 <code>file</code> 指定的文件中
<code>-v[v]</code>	输出所有系统调用参数的详细模式

在理解系统调用参数时，`strace` 是个非常好的工具（如果你使用 `-v/-vv` 参数的话，它就更强大了）。不过截至本书编写时，`strace` 既没有 for Android 的版本，也没有兼容 ARM64 的版本。本书官方网站中提供了一个 `jtrace` 工具<sup>1</sup>，它是 `strace` 的一个克隆版，顺便还解决了其中的一些 bug。

## 本章小结

本章重点讨论了 `/proc` 伪文件系统的使用方法（特别是与每个进程对应的 `/proc/pid` 子目录和与每个线程对应的 `/proc/pid/task/tid` 子目录），其中提供了大量信息，以及让你能执行强大的原生级的进程调试和跟踪的功能。本章中演示的各个方法一样可以应用到主流版本的 Linux 系统上，因为 `procfs` 就是完整的 Linux 内核的一个组成部分。

## 参考文献

文件路径	所提供的信息
<code>/proc/pid/fd</code> <code>/proc/pid/fdinfo</code>	关于进程所打开的文件描述符的信息
<code>/proc/pid/maps</code>	以 <code>mapped</code> 和匿名内存段列表的形式，给出了进程的内存地址空间布局

1 遗憾的是，本书官网上并没有这个工具的下载链接，我（译者）已经发邮件问作者了，作者答应近期给出该工具的下载链接。——译者注

续表

文件路径	所提供的信息
/proc/pid/smaps	和/proc/pid/smaps 的输出类似，同时还给出了各个内存段的实时使用情况
/proc/pid/status	从进程或线程控制块（内核中的 task_struct）那里获取的信息

[1] [www.kernel.org/doc/Documentation/filesystems/proc.txt](http://www.kernel.org/doc/Documentation/filesystems/proc.txt) 描述 procfs 文件系统中各个文件的文档。

## 第 8 章

# Android 安全性

和其他一些方面一样, Android 是依靠底层的 Linux 基础设施来实现基本安全需求的。但是, 对于大部分 Android 应用来说, 它们还受到了由 Dalvik 虚拟机执行的更高一层的安全保护。因此 Android 的安全性就是由虚拟机层和原生代码层这两个层面共同提供的。本章要深入讨论的就是这一内容。

本章开头部分会简要地讨论一下威胁模型——这是安全专家用来分析能够对移动设备安全性造成危害的各种可能的攻击向量和威胁的一种方法。除了台式机安全中需要面临的所有“传统的”威胁以外, 在移动安全中, 至少还要再考虑恶意 App 和小偷这两种潜在的威胁。

接下来, 我们会去讨论 Linux 用户模型, 以及 Android 对它做的改造。我们先讨论“纯”Linux 的权限, 然后再来讨论 Android 中把 ID 用在 App 和组成员上的这一漂亮的做法。再接下来, 我们会重点讲一下权能 (capability), 这是一种常常被忽视的 Linux 特性, 在 Android 中广泛地使用了这一特性, 以克服经典 (安全) 模型中只使用一个全能的 root 账户, 而产生的一些问题。然后我们会讨论 SELinux——这是一个在 Android 4.3 中引入, 并在 4.4 中被强制启用的 MAC (强制访问控制, Mandatory Access Control) 框架。最后, 我们还会简单地讨论一下对抗代码注入攻击的各种保护手段以及应用安全沙箱等问题。

在 Dalvik 虚拟机层上, 我们认为 Dalvik 虚拟机和包管理器执行的是一个简单却非常有效的权限模型, 而这一模型中的权限设置与 Linux 层上的权限的对应关系也是简单且有效的。不过到此为止, 不论是 Linux 层上的还是 Dalvik 虚拟机层上的权限, 我们讨论的都还是应用层 (application level) 上的安全。

所以, 接下来我们要考虑的就是用户层的安全: 通过锁屏机制防止设备被他人使用。除了简单的 PIN 码和图形锁之外, 设备屏幕加/解锁的方式有许多创新, 加/解锁时甚至还可以使用生物特征识别技术。从 JB 版开始, Android 允许在一台设备上同时存在多个用户账户, 这些用



户可以拥有各自独立的私人数据，各自安装不同的应用且互不影响。所以在这一章里也会提及多用户的实现方式。

再接下来，我们要去讨论 Android 中使用的加密机制。我们从密钥管理讲起，解释了 keystore 服务的内部工作机理，以及设备中存放证书的方式。在这之后，我们讨论 Android（Honeycomb 版中引入）的存储加密特性（storage encryption feature），并借助 Linux 的 dm-verity 特性（KitKat 版中引入的），验证文件系统中的数据有无遭到修改的做法。

最后要重点讲述的是（当然，排在最后并不意味着它最不重要）：root 移动设备，不讨论这个话题，任何对安全这一主题的讨论都将是不完整的。root 会给高级用户带来巨大的好处（这也是 Android 在黑客和军火制造商圈子里极度流行的原因之一），但很遗憾，它同时也会给应用和系统安全带来极为严重的影响。我们会详细地讨论和对比两种主要的 root 方式——在设备启动环节中 root 和 “一键 root”。

## 8.1 移动安全威胁建模

考察一下黑客技术的进化史，我们可以看到这样一条进化主线：最初，主要的攻击目标是服务器。（那时）黑掉服务器比黑掉一台个人电脑要容易很多——因为服务器是一直连在网上的 [黑话里甚至把它们称为“呆鹅”（sitting duck）]，而个人电脑上网则是断断续续的——就算它连在网上的时候，用的也是一只超慢的“猫”（modem）。

后来，随着高速网络的接入和本地局域网的兴起，一夜之间，互联网上冒出来了大量的易于攻击的目标。因为和服务器相比，个人电脑的安全态势实在是差得太多了。不安全的默认设置和作为一个过于用户友好的（复杂的）操作系统，使得 Windows 成为孵化黑客的温床，并带来了一大波蠕虫和恶意软件。

### 攻击向量

移动设备在某些方面和个人电脑有些类似，但是在面对的威胁方面却是完全不同的。与后者不同的是：移动设备的高度便携性使之要面对更多的风险，比如它们可能会被无意间放错地方，或者不小心被偷。计算机安全人员可以对个人电脑采取的一些保护措施 [比如给放计算机的房间加锁或者使用密钥（口令卡）保护]，使得攻击者必须先搞定物理防护才能进行进一步的攻击，而对于移动设备来说，这种安保措施是根本无法实施的。

这还只是一半：与个人电脑相比，移动设备更加私人化——这也使它更有可能会记录用户的个人隐私信息，这就使它们成为黑客眼里更加有利可图的攻击目标。攻击牟利的方式也发生

了变化——以前黑客追求的是能够远程完全控制被攻击目标（黑客的行话把这叫“pwning”）。而现在他们经常只需要能够获取用户数据，并在互联网上有一个一直在线的信息接收端（比如，把偷到的数据传到一台服务器上）就够了。

## 不靠谱的 App

移动设备的主要攻击向量来自它的内部——不靠谱的 App。用户总是想安装更多的 App，让手机的功能变得更强大。但如果其中哪个应用中出现了错误的行为，或者干脆就是个有意诱使你安装的恶意应用，它就能试图访问用户数据，甚至是使用手机通信的功能（比如发送诈骗短信）牟利。一般这类攻击会被归类为“本地提权”（local privilege escalation）攻击——因为（攻击发生时）App 已经被安装并运行在移动设备中了，也就是已经在“本地”（local）了，但是它只拥有一些受限制的权限，并想要提升自己的权限。

为了防止出现本地提权，Android 就得把 App 当贼防着。在默认情况下，应用将只被赋予最小权限集中的权限，除了这个最小权限集之外，所有的权限都是被禁止的。特别是：这个最小权限集中是不包括任何可能是敏感的权限的——哪怕这是个非常重要的权限也不行。举个例子来说吧，访问网络的权限可能会被恶意地用来把设备中的信息传出去，因此它就不在最小权限集中。所以，当应用需要任何一个不在最小权限集中的权限时，它就必须在 manifest 文件里显式地声明，要求系统赋予它这个权限。每个应用都会被赋予一个它自己独有的 UID——用来把它和其他应用隔离开来，而且不用说，对于 App 来说，拥有 root 权限更是连门都没有。

从 JellyBean 版开始，通过引入 SELinux，Android 对应用的限制又上了一个新台阶——使用一个强制实施的访问控制框架，有效地把所有的进程置于沙箱里，将其束缚在可信区域内。到了 Android Lollipop 版中，这些框架又被进一步扩展，已经能支持对包（package）进行限制了。

不过光有这些还不够——Android 还必须保护它自己。因为即使一个 App 确实只拥有很少的一些权限，它也可以试图攻击它所在系统中的一些存在安全漏洞的组件。这也不是没有先例，利用这些安全漏洞，进而控制操作系统中拥有更多权限的组件——特别是以 root 权限运行的组件——通过应用的行为，执行某一操作是完全有可能的。由于 Android 框架的实现是由极其大量的代码组成的，更底层的 Linux 内核中的代码更多，这确实是个很严重的威胁。而许多已知的安全漏洞事实上也是使用这个方法完成提权的。

## 不靠谱的用户

很难想象，安全研究人员竟然把移动设备的使用者视为一个现实的威胁（尽管好像 iOS 在某程度上确实也是这么做的）。不过因为手机很容易被偷掉——这让“到底谁才是真正的合法用

户”这个概念变得模糊了起来。故而系统也必须随时保持安全性，特别是在脱离用户的掌控时，尤为如此。

第一道防线是屏幕解锁密码，它必须在较安全的用户认证需求和易于使用（并能快速）解锁之间取得平衡。毕竟，如果每次点亮屏幕时都要输入一个由不同大小写字母组成，长度超过 20 个字符的密码的话，我想想都要晕过去了！所以这就需要用户来确定到底什么样的安全认证是“可接受的”——因此，选择怎样的认证机制以及屏幕在多长时间里没有操作就会被锁定，这些都要交由用户来决定。

Android 中还引入了人脸解锁的这一快速（尽管不怎么安全的）屏幕解锁方式，同时（从 Lollipop 版开始）也紧跟 iOS，开始内置支持指纹认证。Lollipop 版中还加入了使用设备附近的已经成功（通过蓝牙）配对过的设备（通常这个已成功配对过的设备是一个 Android 可穿戴设备）进行解锁的功能。

因为手机或平板电脑在被盗之后，也有可能被关机或重启。所以 Android 也必须保证启动过程是安全的。否则，犯罪分子就会修改 Boot Loader，并用另一个较不安全的配置重启手机/平板电脑。这也就是为什么 Android 设备默认都会对 Boot Loader 加锁，并且一旦 Boot Loader 被解锁，整个/data 分区中的数据就会马上被擦除掉的原因。

最后，用户的数据也应该被加密——再说一遍，一个黑客高手在解锁手机/平板电脑之后，可以直接从底层访问闪存中的二进制数据。最早在 Honeycomb 系统中，Android 就已经提供加密功能了，但直到 Lollipop 版，它才被设为默认启用——又一次晚于 iOS 了。加密的密钥不应该存放在手机里——哪怕是存储在加密存储区域里也不行！为了使可行性最大化，Android 选择从用户的屏幕解锁密码中推算出加密的密钥。

## 远程代码注入

最后，但并非最不重要（如果上面说的这些还不算太糟的话）的，移动设备仍然还要面临和服务器和桌面电脑一样的威胁——远程代码注入。这类安全漏洞既然能黑掉桌面电脑，也一样能黑掉移动设备，只不过现在攻击者攻击的是连上了互联网的移动目标设备。目标设备既可以是被随机“黑掉的”（比如通过恶意垃圾邮件，或恶意的飘窗），也可以是针对某个特定对象的（通常使用社工邮件）被“黑掉”。

Webkit，这个 Android 浏览器或 WebView 的基础，已经被证明存在许多漏洞。这些漏洞通常和恶意的 HTML、CSS、JavaScript 或者混用这三种代码的恶意代码联合使用。谷歌现在已经把 Android 的默认浏览器改成 Chrome 了，不过由于在这个使用非常频繁的组件中出现安全漏洞的可能性是如此之高，以至于在 Lollipop 系统中，对 Chrome 是单独进行版本验证和自动升级



的，而不是和操作系统的其余部分一起等待谷歌慢悠悠地发布的系统升级补丁的。

值得注意的是，在系统启动阶段也会有代码注入问题出现。这时，一个漏洞就能达到类似 Boot Loader 已经被解锁了的效果——也就是说，可以实现把系统启动到任意一种配置状态，且不擦除/data 分区中的数据的效果，因此也就会导致用户数据泄露。

2015 年 7 月，曝出了一个可能是 Android 历史上最严重的漏洞——Joshua Drake（著名安全研究者，并且是 *Android's Hacker Handbook*<sup>1</sup> 一书的作者）发现了一个位于 StageFright 核心库中的漏洞。这个漏洞是通过彩信做到的，即向一台 Android 设备发送一条带视频的恶意彩信后，只要用户一看到彩信就会自动触发远程代码执行 [因为默认的短信/彩信查看器 (Hangouts) 会自动预览这条彩信，从而引爆手机中的漏洞]，这甚至都不需要用户做任何交互操作。尽管这个漏洞并不能让攻击者拿到 root 权限，但是配合另一个由 KEEN team<sup>2</sup> 发现，并在不久之后的 BlackHat 2015 中公开的漏洞利用代码，就能在 Linux 内核中完成本地提权，并 root 整个系统。这个漏洞也被用在“pingpong”<sup>3</sup> root 工具里（这个漏洞也是用它出现的位置 IPPROTO\_ICMP socket 命名的）。

## 攻击之道

到目前为止，我们讨论的这些攻击向量都是黑客使用的主要攻击手段。但一次成功的攻击往往都不是一击得手的，而是通常由多个精心策划、协同一致的攻击步骤共同组成的。图 8-1 演示了一个稍作简化了的攻击实施流程。

如图 8-1 所示，Android 的复杂性给安全防御也带来了不少麻烦——它的受攻击面实在是太多了。Android 中的漏洞可能源自多个不同的地方，它们是：

- **源自 Android**——这些漏洞是来自于 AOSP 本身的。它既可能位于框架的代码中，也可能位于更底层的系统守护进程中。Android 中的大部分守护进程都已经被重写，使它们无需拥有 root 权限就能正常工作，所以大部分守护进程只得到了一个系统 AID。只有很少的几个守护进程（比如 vold 和 lmkd）还在以 root 权限运行。不过从 Android L 版开始，这些进程也受到了 SELinux profile 的制约（详见本章之后的讨论），以增强 Android 的安全性。不过，一旦出现这类漏洞，受影响的将是所有厂商生产的所有设备。

1 中文版为《Android 安全攻防权威指南》，诸葛建伟、杨坤、肖梓航译，人民邮电出版社，2015 年 3 月出版。——译者注

2 现为腾讯科恩实验室，该实验室的负责人是吴石老师。——译者注

3 其实这个漏洞最初就是放在 kingroot 工具里使用的，译者本人还参与了其中部分逆向工程手段的测试工作，不会有错。——译者注

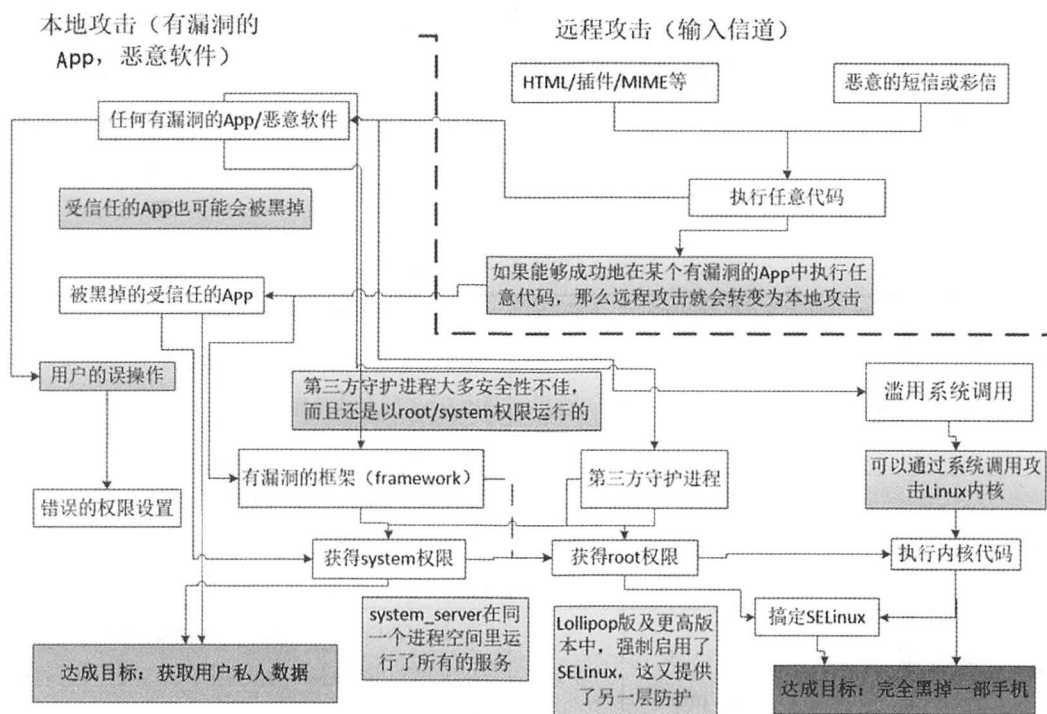


图 8-1 对 Android 系统攻击时所涉及各个步骤相互间关系的粗略示意图

- **源自厂商**——这些漏洞都是出现在厂商（或运营商）在设备中新增的代码或服务（比如第 2 章中讨论的厂商增加的守护进程或预装的 App）中的。这些问题没出在 Android 自身代码中，而是出在那些额外增加进来的组件中，而且这些组件常常还是以超出自身所需的权限运行，又没有足够的安全防护的。
- **源自第三方**——记得我们说过，Android 中使用了大量的其他项目，它们为 Android 提供了许多原生库（native library）以及系统中的守护进程（比如 wpa\_supplicant、mdnsd、racoon 等）。这一做法当然也就引入了海量的代码——而过去的经验表明，这些代码中很可能存在安全漏洞。
- **源自 Linux**——Android 非常依赖于它的 Linux 基础，但就像安全圈的行话所说的那样“信赖往往会变成负担”。Android 不光是（几乎一字不改地）使用了 Linux 的内核，而且还好像是命中注定的那般，在大部分关键组件上照抄了 Linux。因此，在 Linux 内核中发现的任何漏洞，几乎都可以用来黑掉 Android 设备。

*Android Hacker's Handbook*<sup>1</sup>一书中给出了许多 Android 系统中已经公开的漏洞，同时它也是一本很好的参考书籍。但是和其他大多数书籍一样，那本书里讲述的漏洞都已经是“旧闻”了，而且它们都已经被修补了。不过“新闻”还是在不断地涌现出来——每年都会发现越来越多的安全漏洞，就像有一个死循环不断地在产生安全漏洞似的。表 8-1 中列出了从最早的 Android 漏洞一直到目前为止出现过的一些漏洞，并把它们在图 8-1 中的位置一一对号入座地标示出来。

表 8-1 Android 历史上一些最重要的漏洞

CVE/Name	受影响的版本	漏洞源自	攻击向量	造成的影响
CVE-2012-6636	1.0-4.1	第三方	远程	WebView 组件没能安全地处理某些 JavaScript 代码，导致系统默认的浏览器以及其他使用 WebView 的应用会被黑掉
CVE-2014-7911	1.0-4.x	Android	本地	luni/src/main/java/java/io/ObjectInputStream.java 中未能正确地反序列化（deserialization）输入的数据，导致可以在本地以 system 身份执行代码
CVE-2014-4322	4.x	厂商 (msm)	本地	高通的 QSEECOM 驱动中没有安全地调用 ioctl(2)，导致在所有使用高通芯片的设备中都会出现一个修改任意内核内存的漏洞
CVE-2014-3153	4.x	Linux	本地	在 Linux 内核中的 futex(2)系统调用中存在一个 bug，该 bug 会导致一个修改任意内核内存的漏洞
WeakSauce	4.x	厂商 (HTC)	本地	Android 系统中的 dmagent 以不安全的方式复制文件，导致设备可以在本地被 root（这个漏洞在本书官网上有更深入的描述）
CVE-2015-1805	4.x 及以上版本	Linux	本地	一个任意内存覆盖漏洞（这个漏洞一直在被到处使用，谷歌直到 2016 年 3 月才出安全补丁）
CVE-2015-3636	4.x and up	Linux	本地	Linux 内核中的一个释放后使用方面的 bug，导致一个任意内存覆盖漏洞（这个漏洞被用在了 pingpong root 工具中）
CVE-2015-3824	4.x and up	Android	远程	stagefright 框架中的一个漏洞，使得攻击者可以通过一条彩信或一个视频文件，在 mediaserver 进程中执行任意代码

1 该书中文版为《Android 安全攻防权威指南》，诸葛建伟、杨坤、肖梓航译，人民邮电出版社，2015 年 3 月出版。——译者注



续表

CVE/Name	受影响的版本	漏洞源自	攻击向量	造成的影响
CVE-2016-0728	4.4 及以上版本	Linux	本地	keyring 实现代码中的一个整型溢出漏洞，能用来 root 设备（5.0 版中使用 SELinux 缓解了这个问题）
CVE-2016-0819	4.4-6.0.1	厂商 (msm)	本地	高通专用驱动中的一些漏洞，会导致任意内核内存覆盖问题
CVE-2016-5195	所有版本	Linux	本地	“DirtyCOW”漏洞能让设备瞬间被 root

请注意，并不能因为某个漏洞已经出了相应的安全补丁，我们就能心安理得地把它搁在一边了——相当一部分厂商会不再支持上市销售超过两年的设备，而且甚至在发布补丁时，他们也是拖拖拉拉的——不论是在 OTA 推送的更新，还是在需要用户手工下载的补丁时都是如此。此外，有些漏洞甚至可能暂时还没有补丁（特别是 Stagefright 中的那个漏洞和 CVE-2015-1805——尤其是后者，对应的安全补丁在它公开了近一年后才出现！）。

Android 的防御之道

在安全领域中，在两种不同的基础上搭建系统，并不会使系统变得更安全。恰恰相反，事实上这两个基础中只要有任意一个出现了安全漏洞，就足以让整个系统被黑掉。还算好，Android 在它不算长的历史中已经学会了这一点，因为它一次又一次地被黑掉过——尽管 Android 的每个版本都有重大的改进。如表 8-1 所示，有时，漏洞出现在 Android 自身的代码中；有时，问题出在底层的 Linux 内核中；还有时，漏洞又在第三方代码里。因此，我们可以得出这样一个结论：Android 的安全性必须在两个世界（Linux 和它自身）中同时被搞定，并且尽可能有效且尽可能安全地把二者结合起来——尽管历史已经一而再，再而三地证明：光这样还是不够的……

8.2 Linux 层上的安全措施

Android 在 Linux 的基底上构建出了一个富框架 (rich framework)，但在它最核心的层面上，还是要靠 Linux 来完成所有操作的。Android 是构建在 Linux 之上的这一事实，也使得它沿用了 Linux 提供的这些安全特性——权限 (permission)、权能 (capability)、SELinux 和其他一些底层安全保护措施。

Android 使用 Linux 权限的方式

Linux 使用的安全模型就是标准 UNIX 的安全模型。这个模型从 40 多年前发布之后，一直

没有做过任何重大的修改，它使用如下这些基本规定。

- **每个用户都有一个数字形式表示的用户 ID：**具体的用户名是什么反而是无关紧要的——尽管有些用户名是系统保留的 [这些用户名是专门分配给配置文件和它们所属的目录的所有者 (owner) 使用的]。两个用户可能会共享同一个用户 ID，但出现这种情况，从系统的角度讲，实际上只是意味着一个用户有两套用户名/密码而已。
- **每个用户都有一个数字形式表示的主组 (primary group) ID：**和用户名一样，用户组的名称也是无关紧要的。另外，有些 GID (组 ID) 也是系统保留的。
- **用户还可以加入其他的组，成为这些组的成员：**传统上，用户加入其他组后，它是这些组中的成员的这一层关系是记录在 `/etc/group` 文件中的。在该文件中会列出：所有组的名称，组的 id 以及所有加入了某个组，但又不是把它作为“主组”的组成员。
- **文件的访问权限由特定用户 (owner)、组 (group) 和“其他人员”(other) 三部分组成：**我们已经相当熟悉“`ls -l`”命令的输出结果了。在这个输出结果中，文件的访问权限被表示为 (owner) 用户、(owner 所在) 组和“其他人员”的访问权限 (读、写和执行) 的组合。不论是文件还是目录，它们的访问权限都是以这种表达能力非常有限的模型表示的。而且 UNIX 也因为这一模型而受到过一些批评。由于这一模型的局限性，我们只能通过创建专门的用户组的方式来规定文件的访问权限。
- **在 UNIX 中，(几乎) 所有的东西都是以文件的形式来访问的：**事实上，访问各种系统资源 [无论是命名 IPC 对象 (named IPC object)、UNIX domain socket 还是硬件设备] 都是如此。这样就有一个必然的推论：系统资源的访问权限也是遵循文件访问权限约定的。换言之，表示相应设备的文件的访问权限的表示方式和普通文件是完全一样的，而且我们也可以像修改普通文件的访问权限一样，用 `chown/chgrp/chmod` 命令修改这些文件的访问权限。
- **UID 0 是无所不能的：**由于检查文件访问权限的具体实现方式，“0”实际上能够在任何情况下通过检查，以访问所有的文件或资源。推论就是：uid 0 (也就是 root 用户) 在系统中拥有绝对的权力。
- **二进制可执行文件 SetUID 或 SetGID 能让程序以另一个用户的权限 (或者加入另一个用户组) 运行：**不用问，用二进制可执行文件 SetUID/SetGID 启动的程序会被自动授予指定的权限。这一机制乍一看好像是个设计缺陷，但它实际上是个让普通用户能够临时切换执行特权操作 [比如，切换某个用户的 uid (`su` 命令) 或者口令 (`password` 命令)] 的系统特性。根据相关定义，这些特权操作只能由 uid 为 0 的用户执行。但 root 用户也可以授权指定的二进制可执行文件能够执行特权操作 (这时的 SetUID 和 SetGID 就分别等

价于“`chmod 4xxx`”命令和“`chmod 2xxx`”命令)。作为一项（被滥用的）预防措施，如果这些（已经被授权了的）二进制可执行文件被复制或移动到了别处，复制/移动过去的文件中的 `setuid` 和 `setgid` 权限标志位会被清零（也就是不再拥有特权了）。

Android 也继承了这一经典模型（这是底层的 Linux 系统免费提供的），而且也很自然地使用了它，只不过（和 Linux 中的用法相比）在用法上有一个小小的、多少有点新颖的区别：在 Android 中，“用户”是分配给各个应用，而不是分配给使用计算机的人的。突然之间，原本用来区分共享同一个 UNIX 服务器的人类用户的方法，也被用在了应用身上，并且也用相同的隔离措施限定了它们可以使用的权限。一个用户不能访问另一个用户的文件，目录或是进程——这种严密的隔离，使我们可以同时并发地运行多个应用，并且使这些应用不会相互影响。Android 的这一做法确实是独一无二的——iOS 中是在同一个 uid（mobile 或 501）下运行所有进程，并依赖于内核强制使用的沙箱来实现进程间的相互隔离的。

当一个应用初次安装时，`PackageManager` 会把一个唯一的用户 ID 分配给它——我们可以马上猜到：这个 ID 也被称为“应用 ID”（application id）。这个 ID 的取值范围在 10000~90000 这个区间内，然后 `bionic`（Android 的 C 运行时库）会自动把这个 ID 映射为 `app_XXX` 或者 `u_XXXX` 这种更方便人类阅读的格式。

Android 并没有完全摒弃 `SetUID`——因为这需要重新编译内核，并做其他一系列修改。不过从 4.3（JellyBean）版开始，`SetUID` 这个二进制可执行文件已经不再被默认安装了，而且在 `mount /data` 分区时也使用了 `nosuid` 参数。

系统定义的 AID

Android 系统保留了数字较小的用户 ID（1000-9999），专供系统使用。这部分 uid 中只有一部分被使用了，相关定义写在 `android_filesystem_config.h` 文件中。表 8-2 中给出了部分 UID 的宏定义以及它们在 Android 中的作用。这些 UID 大多也同时被用作 GID（组 ID，Group ID 的缩写）：通过加入这些次级组，`system_server`、`adb`、`installd` 及其他一些系统进程就能获得这些组对应的系统文件或设备的访问权限——这是一种相当简单，又很有效果的策略。

表 8-2 Android 系统定义的 GID 以及它们默认的组成员

GID	宏定义	组的成员	权 限
1001	AID_RADIO	system_server	/dev/socket/rild [ 无线接口层（Radio Interface Layer）的守护进程]，能够访问 net.*、radio.* 等属性
1002	AID_BLUETOOTH	system_server	蓝牙配置文件



续表

GID	宏定义	组的成员	权 限
1003	AID_GRAPHICS	system_server	/dev/graphics/fb0，这个是当前屏幕的帧缓存 (framebuffer)
1004	AID_INPUT	system_server	/dev/input/*，输入设备的设备节点
1005	AID_AUDIO	system_server	/dev/eac 或其他声卡设备节点。能访问 /data/misc/audio，能读取/data/audio
1006	AID_CAMERA	system_server	访问与摄像头相关的套接字
1007	AID_LOG	system_server	/dev/log/*
1008	AID_COMPASS	system_server	指南针和地理位置服务
1009	AID_MOUNT	system_server	/dev/socket/vold，它的另一端是 VOLUME 守护进程
1010	AID_WIFI	system_server	WiFi 配置文件 (/data/misc/wifi)
1011	AID_ADB	(保留)	为 ADBD 预留，是 /dev/android_adb 的拥有者 (owner)
1012	AID_INSTALL	Installd	一些应用的数据目录的拥有者 (owner)
1013	AID_MEDIA	mediaserver	能够访问由 media.* 服务访问的/data/misc/media
1014	AID_DHCP	dhcpcd	能访问/data/misc/dhcp 能访问 dhcp 相关属性
1015	AID_SDCARD_RW		Group Owner of emulated SDCard
1016	AID_VPN	mtpd raccoon <sup>1</sup>	/data/misc/vpn，/dev/ppp
1017	AID_KEYSTORE	keystore	能访问/data/misc/keystore (系统的 keystore)
1018	AID_USB	system_server	USB 设备
1019	AID_DRM		能访问/data/drm
1020	AID_MDNSR	mdnsd	多播 DNS (Multicast DNS) 和服务发现
1021	AID_GPS		能访问/data/misc/location
1023	AID_MEDIA_RW	sdcard	/data/media 和物理 SD 卡的拥有者 (owner) 组
1024	AID_MTP		能访问 MTP USB 驱动(它和 mtpd 是没有关系的!)
1026	AID_DRMRPC		DRM RPC

1 原文如此，显然是“raccoon”之误。——译者注

续表

GID	宏定义	组的成员	权 限
1027	AID_NFC	com.android.nfc	支持近场通信（NFC，Near Field Communication）： /data/nfc 和 nfc 服务查找
1028	AID_SDCARD_R		能够对扩展存储设备进行读访问
1029	AID_CLAT		CLAT（IPv6/IPv4）
1030	AID_LOOP_RADIO		无线电设备（Loop Radio device）
1031	AID_MEDIA_DRM		DRM 插件。可访问/data/mediadrms
1032	AID_PACKAGE_INFO		读取包元数据信息
1033	AID_SDCARD_PICS		SD 卡中的 PICS 目录
1034	AID_SDCARD_AV		SD 卡中的 Audio/Video 目录
1035	AID_SDCARD_ALL		SD 卡中的所有目录

Android 的系统属性也依赖于 UID 进行访问控制——如第 4 章中所讨论过的，init 的 property\_service socket 是通过各种系统属性的命名空间（namespace）来进行访问限制的。同样的方法也被用来保证 servicemanager [它是所有 IPC（进程间通信）赖以实现的关键服务] 的安全性。尽管 Binder 最终是通过 uid/pid 模型提供安全保障的，但 servicemanager 也能够把众所周知（well known）的服务的名称的查询，限制在指定 uid 范围内——尽管 uid 0 或 SYSTEM 用户总是例外的。直到 KitKat 版（含）为止，上述 GID 都是写死在一个名为“allowed”的数组中的，这个数组的内容如代码列表 8-1 所示。

在引入了 SELinux，并且缓慢但坚定地把它并入了 Android 中之后，这种在源码中写死权限的做法最终被抛弃了。取而代之的是：把它合并到 SELinux 策略设置中——类似的情况也发生在 init 的系统属性安全设置身上。不论如何，有一点是一定要注意到的：AID 并不是 Android 中唯一的安全层：servicemanager 会拒绝不可信的 AID 注册众所周知（well know）的服务名，但正如我们下面将会讨论的那样，Binder 还会对客户端或服务端执行额外的权限检查，而 Dalvik 虚拟机也在另一层上使用了另一套安全措施。

代码列表 8-1 KitKat 版中服务的权限（代码源自 service\_manager.c）

```

/* TODO:
 * These should come from a config file or perhaps be
 * based on some namespace rules of some sort (media
 * uid can register media.*, etc)
 */
static struct {
    unsigned uid;
    const char *name;
} allowed[] = {
    { AID_MEDIA, "media.audio.flinger" },
    { AID_MEDIA, "media.log" },
    { AID_MEDIA, "media.player" },
    { AID_MEDIA, "media.camera" },
    { AID_MEDIA, "media.audio_policy" },
    { AID_DRM, "drm.drmManager" },
    { AID_NFC, "nfc" },
    { AID_BLUETOOTH, "bluetooth" },
    { AID_RADIO, "radio.phone" },
    { AID_RADIO, "radio.sms" },
    { AID_RADIO, "radio.phonesubinfo" },
    { AID_RADIO, "radio.simphonebook" },
    /* TODO: remove after phone services are updated: */
    { AID_RADIO, "phone" },
    { AID_RADIO, "sip" },
    { AID_RADIO, "isms" },
    { AID_RADIO, "iphonesubinfo" },
    { AID_RADIO, "simphonebook" },
    { AID_MEDIA, "common_time.clock" },
    { AID_MEDIA, "common_time.config" },
    { AID_KEYSTORE, "android.security.keystore" },
};
// .....
int svc_can_register(unsigned uid, uint16_t *name)
{
    unsigned n;

    if ((uid == 0) || (uid == AID_SYSTEM)) return 1;

    for (n = 0; n < sizeof(allowed) / sizeof(allowed[0]); n++)
        if ((uid == allowed[n].uid) && str16eq(name, allowed[n].name))
            return 1;

    return 0;
}

```

## 执行严格网络权限控制时使用的 Android GID

当 CONFIG\_PARANOID\_ANDROID 被置为 1 时，Android 内核还能认出在 3000 到 3999 这个区间里的 GID。这时，通过在内核中的套接字处理代码中强制添加 GID 检查的方式，使得与网络访问相关的所有功能都只能被这些 GID 或其组成员使用。注意，netd 可以撤销这些设置，因为它是以 root 身份运行的。表 8-3 中显示的是已知的与网络相关的 AID。



表 8-3 Android 与网络相关的 AID 及它们的组成员

GID	宏定义	组成员	权 限
3001	AID_BT_ADMIN	system_server	创建 AF_BLUETOOTH 套接字
3002	AID_NET_BT	system_server	创建 sco、rfcomm 或 l2cap 套接字
3003	AID_NET_INET	system_server	/dev/socket/dnsproxyd 和 AF_INET[6] (IPv4, IPv6) 套接字
3004	AID_NET_RAW	system_server, mtpd, mdnsd	创建 raw [而非 TCP/UDP 或多播 (multicast)] 套接字
3005	AID_NET_ADMIN	raccoon,mtpd	配置网卡和路由表
3006	AID_NET_BW_STATS	system_server	读取上网流量统计信息
3007	AID_NET_BW_ACCT	system_server	修改上网流量统计信息

## 服务分离

从 JellyBean (4.1) 版开始, Android 引入了“服务分离”(isolated service)这一概念。该特性是某种形式的功能分离(类似于 iOS 的 XPC),它使应用能把它的服务分离到一个完全隔离的区域(即在另一个拥有独立 UID 的进程里)运行。被分离出来的服务使用的是 99000 到 99999 (AID\_ISOLATED\_START 从 AID\_ISOLATED\_END)这个区间里的 UID, servicemanager 会拒绝所有来自这些进程的请求,其结果就是:这些进程不能使用任何系统服务,只能完成一些处理内存中数据的操作。这些服务主要是在 Web 浏览器之类的应用中使用的,而 Chrome 则是使用这一机制的一个典型的例子。在输出结果 8-1 中,分离出来的服务进程的(拥有者/owner) uid 以 u##\_i##这样的格式被标记了出来。

```

shell@htc_m8wl:/ $ ps | grep chrome
u0_a114  4577 384 1178728 118528 ffffffff 4007941c S com.android.chrome
u0_i0    5510 384 1283624 89788 ffffffff 4007941c S com.android.chrome:sandboxed_process0
#
# Pulling the Chrome.apk to the host and dumping its manifest:
#
morpheus@Forge (/tmp)$ /aspt_d xmlltree Chrome.apk AndroidManifest.xml
....
E: service (line=285)
A: android:name(0x01010003)="org.chromium.content.app.SandboxedProcessService0"
A: android:permission(0x01010006)="com.google.android.apps.chrome.permission.CHILD_SERVICE"
A: android:exported(0x01010010)=(type 0x12)0x0
A: android:process(0x01010011)=":sandboxed_process0"
A: android:isolatedProcess(0x010103a9)=(type 0x12)0xffffffff 0xffffffff="true", so isolated
# ... 12 more entries
E: service (line=298)
A: android:name(0x01010003)="org.chromium.content.app.PrivilegedProcessService0"
A: android:permission(0x01010006)="com.google.android.apps.chrome.permission.CHILD_SERVICE"
A: android:exported(0x01010010)=(type 0x12)0x0
A: android:process(0x01010011)=":privileged_process0"
A: android:isolatedProcess(0x010103a9)=(type 0x12)0x0 0x0="false", so not isolated
....

```

输出结果 8-1 从 Chrome 中分离出来的服务

拥有者为 root 用户的进程

就像在 Linux 系统中一样，root 用户（uid 为 0 的用户）是无所不能的，但却远非无所不在的——限制 root 用户的使用是为了贯彻最小权限原则，而且随着 Android 的不断升级，本来已经很少被使用的 root 用户，会被用得就更少。之前有一些 Android 系统的 root 工具就是利用了 root 所拥有的进程中的漏洞（特别是 vold，这个进程常被用作这类跳板）来 root 手机/平板电脑的，所以谷歌希望通过减少 root 所拥有的进程的数目的方式，缩小 Android 系统的受攻击面。installld 就是这样一个例子：这个进程以前是拥有 root 权限的，不过从 JellyBean 开始，它的 root 权限就被去掉了。

不过把全部 root 所拥有的进程的 root 权限都去掉好像也是不可能的：至少，init 进程必须拥有 root 权限，同样 Zygote 进程也要有 root 权限（Zygote 进程要 fork() 出 uid 不一样的子进程，这一点只有 uid 0 才能做到）。你可以用下面这条命令来列出你的手机中所有 root 所拥有的进程：

```
ps | grep ^root | grep -v "2"
```

其中，“grep -v”命令的作用是忽略掉所有 PPID 为 2 的内核线程。

表 8-5 中列出的是，在 KitKat 系统中默认以 root 权限运行的服务（不过请注意，在你自己的手机上，root 所拥有的进程可能会更多些，因为手机的生产商还可能会往里加上一些以 root 权限运行的服务）。

表 8-4 Android 中仍在以 root 权限运行的服务

服 务	为什么需要 root 权限
init	它必须拥有系统中的 root 权限，不然就没法启动其他进程了——这可能也是因为它还是 pid 为 1 的那个进程
ueventd (init)	这个进程最简单的操作都需要有 root 权限
healthd	这个进程最简单的操作都需要有 root 权限
zygote[64]	在 fork 时，Zygote 需要有 root 权限才能调用 setuid()，把 fork 出来的进程的 AID 改成被加载的 APK 的 AID，此外，还需要为 system_server 保留相应的权能
debugger[64]	在生成 tombstone <sup>1</sup> 时，debugger[64]需要有 root 权限才能调用 ptrace(2)，以读取被杀掉进程内存中的信息

1 类似一个微型的崩溃转储，详见第 5 章 “debuggerd<sup>[64]</sup>” 一节。——译者注

续表

服 务	为什么需要 root 权限
adb	开发者有时可能需要合法地以 root 权限访问设备： 如果 ro.debuggable 标志位为 0，或者 ro.secure 标志位为 1，则系统将会信任 ADB，并立即给 adb shell 赋予 root 权限
vold	mount 或 unmount 文件系统等
netd	配置网卡，设置 IP 地址或以 DHCP 获取 IP 地址等
lmkd	调整 OOM 设置，此外，这个进程还需要在必要时杀掉进程（因此也需要 root 权限）

另外，在第 2 章里我们也提到过，制造商塞进系统的那些二进制可执行文件也会在很大程度上扩大 Android 系统的受攻击面——尤其是在它们还是以 root 权限运行时更是这样。特别是：由于 AOSP 是开源的，所以我们可以很方便地全方位地分析它的安全性；但厂商塞进来的这些二进制可执行文件却都是闭源的，而且为了实现某些功能，有些厂商还会牺牲安全性！这就使问题更加严重了。所以，如果你听说某个漏洞只能在特定型号的手机（比如，HTC One M8）上，而不是某个版本的所有 Android 系统上使用时，这个漏洞的根子很可能就在厂商塞进来的二进制可执行文件里。

我们可以预见到，最终，Android 只会让那些必须拥有 root 权限的服务保有 root 权限，而其他的服务都会步 installd 的后尘。为了实现这一点，Android 不得不更加倚重另一种重要的 Linux 安全特性——权能。

### Linux 权能

最初作为 POSIX.1e 草案的一部分（并且这也意味着被并入所有 UNIX 系统的标准之中），权能（capability）早在 2.2 版起，就已经被 Linux 内核所采纳了。尽管 POSIX 草案最终流产了，但权能仍在 Linux 中得以实现，并从此得到了不断扩充和改进。尽管 Linux 的各个发行版中并不总是会使用权能，但在 Android 中却广泛地使用着它。

权能背后的设计思想是：打破只有一个强大的 root 用户，“要么什么都能做/要么什么都不能做”（all-or-nothing）的模型——root 用户是上天下海无所不能的，而所有其他的用户实际上都是没有啥权力的。有鉴于此，如果一个用户需要做一些特权操作，唯一的标准做法就是先通过 SetUID 变成 uid 0，执行相关操作，然后再交出超级用户权限，回归到原来非特权用户的身份。甚至是只需执行一些十分简单的操作（比如设置系统时间，绑定某些专用的（小于 1024 的）网络端口，mount 某些文件系统等）时，也必须这般办理。这样做导致的一个结果就是：在 UNIX 系统中，通常都有大量能 SetUID 的二进制可执行文件。



如果一个能 SetUID 的二进制可执行文件是可信的，那么至少在理论上这一模型还是能有效工作的。不过在实践中，SetUID 天生就有一些安全隐患：如果一个能 SetUID 的二进制可执行文件中被发现存在安全漏洞，攻击者就能借助它取得 root 权限。常见的漏洞利用方式包括：利用符号链接（symlink）、竞争条件（race condition）（借助存在漏洞的程序覆盖系统的配置文件）和代码注入（导致存在漏洞的程序执行一个拥有 root 权限的 shell——后文中我们也使用术语“shellcode”指代“代码注入”）。

权能提供了一个能解决这一问题的解决方案——把 root 权限拆分成若干种各不相交的子权限，每种权限用 bitmask 中的一个 bit 位来表示。通过对这个 bitmask 中的各个标志位进行设置，可以让相关程序能够执行与指定权限对应的特权操作，同时又将该程序的权限限制在被授予的权限范围内，使之不能执行其他类型的特权操作。这使得权能成了“最小权限原则”的一种实现方式（“最小权限”是一条安全原则，即任一应用程序或用户不得拥有超出执行正常操作所必须的权限以外的任何权限）。我们可以从图 8-2 中看到权能的逻辑视图。

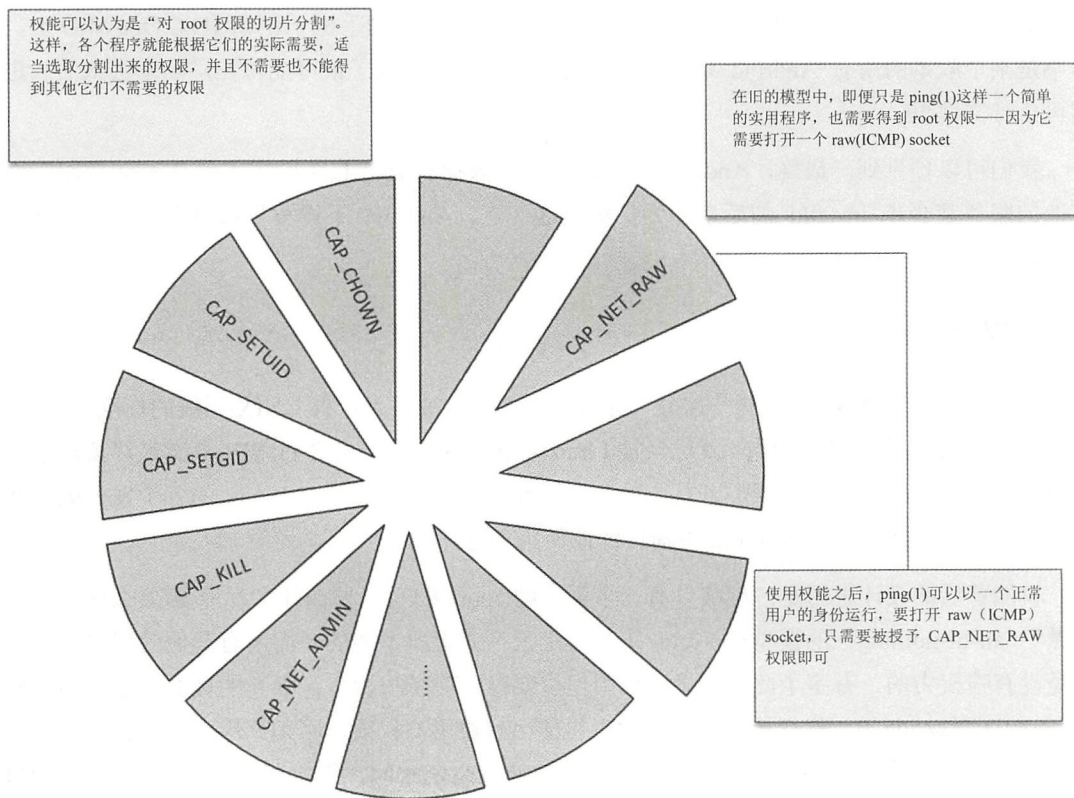


图 8-2 权能的逻辑视图

把权限限制在一个子集中——只赋予那些绝对必须的权限，同时保留其他的权限，极大地增强了安全性。即便给定的应用程序或用户最终是恶意的（或者因为代码注入而执行了不该执行的操作），危害的范围也会被限制在一定的范围之内。权能就像是一个沙箱，允许应用执行设计所需的操作——同时又防止它为所欲为，危害系统安全。事实上，权能还有个很好的副作用：我们能在 root 账号的使用者并不是完全值得信任时，使用权能限制 root 用户本身的行为。

init 进程仍然会以 root 权限启动大部分 Android 服务，在刚开始启动时这些服务进程的权能标志位确实是全部为 true 的（即，bitmask 为 0xffffffffffff）。但是在这些服务进程开始真正做些什么之前，它们会把自己的权能降下来，只保留它们必须使用的那部分权能。installld 就是这样一个坚持最小权限原则的很好的例子：它会确保把除了安装包（package）所必需的权限之外的所有权限都交出去（见代码清单 8-2）。

代码清单 8-2 installld 进程交还部分权能的代码

```
static void drop_privileges() {
    // Ask the kernel to retain capabilities, since we setgid/setuid next
    if (prctl(PR_SET_KEEPCAPS, 1) < 0) {
        ALOGE("prctl(PR_SET_KEEPCAPS) failed: %s\n", strerror(errno));
        exit(1);
    }

    // Switch to gid 1012
    if (setgid(AID_INSTALL) < 0) {
        ALOGE("setgid() can't drop privileges; exiting.\n");
        exit(1);
    }

    // Switch to uid 1012
    if (setuid(AID_INSTALL) < 0) {
        ALOGE("setuid() can't drop privileges; exiting.\n");
        exit(1);
    }

    struct __user_cap_header_struct capheader;
    struct __user_cap_data_struct capdata[2];
    memset(&capheader, 0, sizeof(capheader));
    memset(&capdata, 0, sizeof(capdata));
    capheader.version = _LINUX_CAPABILITY_VERSION_3;
    capheader.pid = 0;

    // Request CAP_DAC_OVERRIDE to bypass directory permissions
    // Request CAP_CHOWN to change ownership of files and directories
    // Request CAP_SET[UG]ID to change identity
    capdata[CAP_TO_INDEX(CAP_DAC_OVERRIDE)].permitted |= CAP_TO_MASK(CAP_DAC_OVERRIDE);
    capdata[CAP_TO_INDEX(CAP_CHOWN)].permitted        |= CAP_TO_MASK(CAP_CHOWN);
    capdata[CAP_TO_INDEX(CAP_SETUID)].permitted        |= CAP_TO_MASK(CAP_SETUID);
    capdata[CAP_TO_INDEX(CAP_SETGID)].permitted        |= CAP_TO_MASK(CAP_SETGID);

    capdata[0].effective = capdata[0].permitted;
    capdata[1].effective = capdata[1].permitted;
    capdata[0].inheritable = 0;
    capdata[1].inheritable = 0;

    if (capset(&capheader, &capdata[0]) < 0)
        ALOGE("capset failed: %s\n", strerror(errno)); exit(1);
}
```

使用权能最多的用户无疑就是 `system_server` 了。因为尽管这个进程的所有者是 `system`，但在执行许多它的普通操作时，还是需要 `root` 权限的。表 8-5 中列出了各种 Linux 的权能，以及使用这些权能的 Android 进程。

表 8-5 Android 进程使用的 Linux 权能

权能标志位	宏	使用者	权 限
0x01	CAP_CHOWN	installd	更改文件和组的所有者
0x02	CAP_DAC_OVERRIDE	installd	可以任意覆盖文件/目录的访问控制设置
0x20	CAP_KILL	system_server	杀掉和自己不属于同一个 uid 的进程
0x40	CAP_SETGID	installd	允许执行 <code>setuid(2)</code> 、 <code>seteuid(2)</code> 和 <code>setfsuid(2)</code>
0x80	CAP_SETUID	installd	允许执行 <code>setgid(2)</code> 和 <code>setgroups(2)</code>
0x400	CAP_NET_BIND_SERVICE	system_server	绑定小于 1024 的本地端口
0x800	CAP_NET_BROADCAST	system_server	广播/多播
0x1000	CAP_NET_ADMIN	system_server	配置网卡、路由表等
0x2000	CAP_NET_RAW	system_server	使用 raw socket
0x10000	CAP_SYS_MODULE	system_server	插入/删除内核中的模块
0x800000	CAP_SYS_NICE	system_server	设置进程的优先级和进程的亲和性（affinity）
0x1000000	CAP_SYS_RESOURCE	system_server	设置进程的资源限制
0x2000000	CAP_SYS_TIME	system_server	设置实时时钟
0x4000000	CAP_SYS_TTY_CONFIG	system_server	配置/挂起 tty 设备
0x400000000	CAP_SYSLOG	dumpstat	配置记录在 ring buffer 中的内核日志（dmesg）
0x1000000000	CAP_MAC_OVERRIDE	system_server	修改 MAC 策略设置（实际上会被忽视）

注意，表 8-5 中只给出了一部分 Linux 权能（尽管已经很长了）。随着 Linux 和 Android 版本的进一步更新，很可能还会有更多的权能被添加进来。下面这个实验中演示了如何查看各个进程所使用的权能。

### 实验：查看进程使用的权能及已经加入的组

通过查看 `/proc/${PID}/status` 伪文件，可以很方便地查询到 `system_server` 进程（或者其他进程）所使用的权能以及它所加入的各个组，只要把路径 `/proc/${PID}/status` 中的 `${PID}` 替换成要查询的进程的 PID 即可，如输出结果 8-2 所示。



```

root@generic:/ # cat /proc/${SS_PID}/status
Name:   system_server
State:  S (sleeping)
Tgid:   372
Pid:    372
Ppid:   52
TracerPid: 0
Uid:    1000    1000          # AID_SYSTEM
Gid:    1000          # AID_SYSTEM
...     # Note the secondary group memberships, below
Groups: 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1018 1032 3001 3002 3003 3006 3007
...
CapInh: 0000000000000000    # Inherited
CapPrm: 0000000007813c20    # Permitted
CapEff: 0000000007813c20    # Effective
CapBnd: ffffffff00000000    # Bounding set

```

输出结果 8-2 查看 system\_server 所使用的权能，及它已经加入了哪些组

在上面这个输出结果中，我们可以看到四个关于权能的 bitmask：可以被子进程继承的权能、被授予该进程的权能、活跃有效的权能（即那些被授予该进程，而且也是该进程明确需要的权能）以及权能范围（Bounding set）。（Linux 2.6.25 中引入的）权能范围是一个用来限制 capset(2) 作用范围的 bitmask。

通过遍历 /proc 伪文件系统中各个进程的 PID 目录中的 status 伪文件，我们还能挑出所有使用了权能的进程。不过要达到这一目的，我们得敲入几行 shell 脚本，如输出结果 8-3 中所示。

```

root@htc_m8wl:/proc # for p in [0-9]*; do CAP=$(grep CapPrm $p/status) \ # Iterate over all PIDs
do CAP=$(grep CapPrm $p/status) \ # Get permitted capabilities
grep -v -v ffffffff \ # Rule out root processes
grep -v 0000000000000000 \ # Rule out incapable processes
if [[ ! -z $CAP ]]; then \ # If capabilities were found.
grep Name $p/status; \ # Print the process name
echo PID $p - $CAP; \ # PID, and capabilities mask
fi; \
done
Name: system_server
PID 13662 - CapPrm: 0000000007813c20
Name: wpa_supplicant
PID 13907 - CapPrm: 00000000000003000
Name: rildd
PID 368 - CapPrm: 00000000000003000
Name: netmgrd
PID 375 - CapPrm: 00000000000003000
Name: installd
PID 387 - CapPrm: 00000000000000c3
Name: dumpstate
PID 389 - CapPrm: 0000000400000000
Name: dumpstate
PID 390 - CapPrm: 0000000400000000
Name: qseecomd
PID 398 - CapPrm: 0000000000222000
Name: qseecomd # A secondary thread of qseecomd, therefore same capabilities
PID 510 - CapPrm: 0000000000222000

```

输出结果 8-3 使用了权能的进程

从上面这个输出结果所示内容我们可以看到：这些使用了权能的进程也出现在表 8-4 中。请注意，有些厂商也会给它们自己的进程添加额外的权限——比如上面这个输出结果中的

qseecomd 进程就是一例（我是在一台 HTC 手机上完成这个实验的）。

从 JellyBean(4.3 版)开始, Zygote 会去调用 `prctl(PR_CAPSET_DROP)`和 `prctl(PR_SET_NO_NEW_PRIVS)`, 以确保不会有其他权能够添加给它的子进程（也就是用户打开的各种应用）。很可能, 以后 `vold` 和 `netd` 也都会主动降低自己的权限, 只获取部分必须的权能, 而不是像现在这样拥有 `root` 权限。考虑到 `vold` 历史上被曝出过存在多个安全漏洞, 这一做法显得尤为重要。

## SELinux

SELinux, 即安全加固的 Linux (Security Enhanced Linux), 使 Linux 在超越标准 UNIX 的道路上跨出了一大步(见图 8-3)。SELinux 最初由 NSA 开发时, 只是一个补丁集, 后来被合并进了内核主线 (mainline), 以提供一个能够根据预先定义的策略规则 (policy), 对相关操作进行审查限制的强制访问控制 (MAC, Mandatory Access Control) 框架。和权能一样, SELinux 也实现了最小权限原则: 只是粒度更细罢了。通过严格地防止 (进程的) 操作超出预定义的操作边界, SELinux 极大地增强了系统的安全态势。只要进程的行为良好, 就不会有任何问题。但如果进程行为不端 (在大多数情况下, 这可能是因为进程本身就是恶意的, 也可能是因为进程被注入了恶意代码而在正常进程中出现恶意操作), SELinux 就会阻断所有这些越界操作。这一方法非常类似于 iOS 中 (基于 TrustedBSD MAC 框架) 的沙箱——尽管实现方式非常不同。

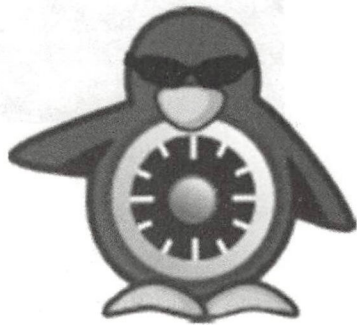


图 8-3 SELinux 的 Logo

尽管已经并入 Linux 很久了 (而且和权能一样, 在 Linux 发行版中并不会总是被默认启用), 但 SELinux 一直到 JellyBean (4.3 版) 才被引入到 Android 中。而且刚开始引入它时的做法也很温和: SELinux 被设为宽容 (permissive) 模式, 在这一模式下, 任何违反策略规则 (policy) 的行为几乎都不会被审计。但随着 KitKat (4.4 版) 的到来, 现在 SELinux 默认被设为: 对一些 Android 服务 (特别是对 `installd`、`netd`、`vold` 和 `zygote`) 启用了强制 (enforcing) 模式, 而对所有其他进程仍使用宽容模式。一般而言, 对每个域 (domain) 先使用宽容模式, 以便在把它设为强制模式之前, 对相关策略规则进行测试, 被认为是一种良好的工程策略。所以有理由认为: 在 Android 的下一个版本中, 强制模式的使用范围会进一步扩大。

Android 中使用的 SELinux 通常被称为 SEAndroid, 这一称谓第一次是在 NSA 的 Smalley 和 Craig 的一篇文章<sup>[1a]</sup>和一个 PPT<sup>[1b]</sup>里提出来的。谷歌公司在 Android 源码站点<sup>[2]</sup>中也提供了一篇基础文档。此外, 作为使用 Linux 内核源码主线制作的发行版之一, RedHat 也是较早采用



SELinux 的 Linux 发行版本，并提供了一篇关于 SELinux 的较为详尽的使用向导手册<sup>[3]</sup>。

SEAndroid 沿用了 SELinux 使用的原则，只是为了能够兼容 Android 特有的特性 [比如系统属性，当然还有 Binder (通过内核 hook)]，对部分内容进行了扩充。三星对 SEAndroid 做了进一步的扩展，并把它作为它们自己的“KNOX”安全平台 (当前最新的是 v2.0 版) 的基础。KNOX (有些人也称之为“obKNOXious”) 自诩采用了更严格的安全策略，使用强制模式，限制所有进程 (init 和内核线程除外) 的行为。在接下来的讨论中，我们把 Android 和 Linux 中都有的那些特性称为“SELinux”，只有那些只出现在 Android 中的特性，才会把它们称为“SEAndroid”。

SELinux 所使用的基本方法是“打标签” (labeling) (事实上它也是大多数 MAC 框架所使用的基本方法)。所谓“标签” (label) 就是为资源 [客体 (object)] 分配一个类型 (type)，或为进程 [主体 (subject)] 分配一个安全域 (security domain)。SELinux 可以据此强制让只有在同一个域中 (即，打了同样的标签) 的进程才能访问对应的资源 [有些 MAC 框架走得更远，即会使资源对于打了不同的标签的进程是不可见的，这有点像 Linux 中命名空间 (namespace) 的概念，不过 SELinux 并没有走这么远]。根据策略规则，域也可以是有限的 (confined)，这时，除了被允许访问的资源之外，进程就不能访问任何其他资源了。策略规则的执行是独立于其他权限层 [比如，文件的 ACL (访问控制列表)] 的。策略也可以允许在某些情况下，给一些已经打了标签的对象重新打新的标签 [relabelto 和 relabelfrom，也被称为域切换 (domain transition)]，在一个可信进程 (比如，Zygote) fork() 出一个不可信的进程 (实际上就是用户安装的应用) 时，这一操作是绝对必要的。

一个 SELinux 标签实际上就是一个 4 元组，也就是一个 user:role:type:level 格式的字符串。被打了同样标签 (也就是处于同一个域中) 的所有进程相互之间是等价的。(目前) SEAndroid 中只定义了 type——换言之，标签应该总是 u:r:domain:s0 这种形式的。截至 KitKat，SEAndroid 策略为每个守护进程都定义了一个独立的域 (即每个守护进程都会有它自己的权限和安全 profile 设置)，而 Android 应用中的类也有对应的域，表 8-6 中列出的就是这些域。

表 8-6 Android 4.4 中各个应用类对应的域

标签 (域)	APP	限 制
u:r:kernel:s0	保留给内核线程	没有任何限制 (上帝模式)
u:r:isolated_app:s0	独立的进程	以读和写权限，预先连接匿名 UNIX socket
u:r:media_app:s0	media key 签名的应用	允许访问网络
u:r:platform_app:s0	platform key 签名的应用	
u:r:shared_app:s0	shared key 签名的应用	



续表

标签（域）	APP	限制
u:r:release_app:s0	release key 签名的应用	
u:r:untrusted_app:s0	其他所有进程	能访问 ASEC、SD 卡、TCP/UDP socket、PTY

表 8-6 中提到的各个 key 都是定义在/system/etc/security/mac\_permissions.txt 文件中的，这个文件是 MMAC（中间件 MAC，middleware MAC）实现的一部分：包管理器（Package Manager）能够识别出用这些 key 签名的 App，并给它打上相应的标签（通过调用 SELinuxMMAC.assignSeinfoValue）。该操作是在执行包扫描（package scanning，这是包安装过程的一部分，详见第 2 本的相关讨论）的过程中完成的。注意，这里使用的术语“中间件”（middleware）指的是：用 Android 的系统组件，在用户模式下严格执行 SELinux 安全策略的实现方式。

所有的 xxx\_app 域都是继承自最基本的 domainapp 这个域的。这个域只允许执行一些最基本的行为，其中包括使用 Binder、与 Zygote 通信、surfaceflinger 等。在 AOSP 的 external/sepolicy 目录下，我们可以找到记录所有域的详细定义的各个.te（type enforcement）文件。这些文件中使用的是一种混合了关键字（keyword）和宏（来自 temacros）的语法，定义了所有在这个域中允许或不允许执行的操作，如代码清单 8-3 所示。

代码清单 8-3 .te 文件一例（debugger.te）

```
# debugger interface
type debuggerd, domain;
permissive debuggerd;
type debuggerd_exec, exec_type, file_type;

init_daemon_domain(debuggerd) # force automatic transition when init spawns us
unconfined_domain(debuggerd) # Leaves debuggerd unconfined, at present
relabelto_domain(debuggerd) # Allow domain transition to this domain
allow debuggerd tombstone_data_file:dir relabelto; # For tombstone files
```

external/sepolicy 目录中的这些文件形成了一个基线（baseline），所有设备中使用的 SELinux 策略规则设置都是继承自它的。谷歌并不鼓励厂商修改 external/sepolicy 目录中的这些文件，而是鼓励在它们的 BoardConfig.mk 文件中通过添加四个特定变量的方式，自定义相关的 SELinux 策略规则。在这四个变量中，BOARD\_SEPOLICY\_REPLACE、BOARD\_SEPOLICY\_UNION、BOARD\_SEPOLICY\_IGNORE 分别是用来替代、添加和忽略策略规则中的指定.te 文件的，而 BOARD\_SEPOLICY\_DIRS 则是用来添加搜索路径，使应用能够访问存有应用自己的.te 文件的目录的。这样做能够减少因为文件中出现错误，而在无意间改动了 SELinux 策略规则，进而产生安全漏洞的风险。在这个目录中还含有一个名为 mac\_permissions.xml 的模板，可以通过 insertkeys.py 获取 keys.conf 中定义的 key 后，得到最终的 mac\_permissions.xml。

移动设备上存放的 SELinux 策略规则是将上述文件编译整合之后得到的/sepolicy 文件——这是一个存放在根（root）文件系统中的二进制数据文件。这一做法进一步提升了安全性，因为根文件系统是由 initramfs mount 上来的，而 initramfs 则是有数字签名保护的（因而也就可以被认为是不可修改的）系统镜像的一部分。而编译的目的也只是优化而已，所以/sepolicy 文件是很容易反编译的，如下面这个实验所示。/sepolicy 这个文件可以用/sys/fs/selinux 来加载（尽管在大多数情况下，init 是通过 libselinux.so 这个库中的代码来执行这一操作的）。

### 实验：反编译 Android 中的/sepolicy 文件

如果你有一台 Linux 计算机，那你就可以使用 sedispol 命令，来反编译/sepolicy 文件。sedispol 命令是 checkpolicy 包的一个组成部分，我假设你使用的是 Fedora 或者类似的系统，要是你还没装这个包的话，那么你需要先去下载和安装这个包，见输出结果 8-4。

```
root@Forge (~)# yum install checkpolicy
Loaded plugins: langpacks, refresh-packagekit
--> Running transaction check
--> Package checkpolicy.x86_64 0:2.1.12-3.fc19 will be installed
--> Finished Dependency Resolution
..
Installed:
  checkpolicy.x86_64 0:2.1.12-3.fc19
root@Forge (~)# rpm -ql checkpolicy
/usr/bin/checkmodule
/usr/bin/checkpolicy
/usr/bin/sedismod
/usr/bin/sedispol # This is the policy disassembler
/usr/share/man/man8/checkmodule.8.gz
/usr/share/man/man8/checkpolicy.8.gz
```

输出结果 8-4 下载和安装 checkpolicy 包

安装好了这个工具之后，你还要把/sepolicy 这个策略文件从移动设备上 pull 到计算机中，然后才能开始对它做反编译（这个命令是个交互式命令）。

policy 通常是定义在/sepolicy 中的一个项，你也可以通过 sysfs 文件系统中的相关命令主动加载它。在/sys/fs/selinux/目录中存有不少有意思的能够用来配置（实际上也可以用来禁用）SELinux 的工具。你所要做的，不过是根据输出结果 8-5 中的做法，完成操作即可。

```

root@htc_m8w1:/ # ls -l /sys/fs/selinux/policy /sepolicy
-r----- root      root      74982 1970-01-01 01:00 policy
-rw-r--r-- root      root      74982 1970-01-01 01:00 sepolicy
root@htc_m8w1:/ # cp /sys/fs/selinux/policy /data/local/tmp
root@htc_m8w1:/ # chmod 666 /data/local/tmp/sepolicy
#
# Back on the host (as any user)
#
morpheus@Forge (~)$ adb pull /data/local/tmp/sepolicy
2750 KB/s (74982 bytes in 0.026s)
morpheus@Forge (~)$ sedispol sepolicy
Reading policy...
libsepol.policydb_index_others: security: 1 users, 2 roles, 287 types, 1 bools
libsepol.policydb_index_others: security: 1 sens, 1024 cats
libsepol.policydb_index_others: security: 84 classes, 1333 rules, 1 cond rules
binary policy file loaded

Select a command:
1) display unconditional AVTAB
...
Command ('m' for menu): 1
allow gemud installld : udp_socket { ioctl read write create getattr setattr lock relabelfrom
relabelto append bind connect listen accept getopt setopt shutdown recvfrom sendto recv_msg
send_msg name bind node bind };
allow system installld : udp_socket { ioctl read write create getattr setattr lock relabelfrom
relabelto append bind connect listen accept getopt setopt shutdown recvfrom sendto recv_msg
send_msg name bind node bind };
allow keystore ctl_dumpstate_prop : property_service { set };
allow keystore ping : peer { recv };
... # Probably more output than your terminal can buffer - consider "f" for file output..

```

输出结果 8-5 反编译/反汇编当前活跃的策略规则

讲完了策略规则，接下来要讲的是 SELinux 如何给资源打标签<sup>1</sup>、设置上下文（context）。SELinux 能够识别的资源就是 Linux 文件对象（包括 socket、设备节点以及其他可以用文件来表示的对象），而 SEAndroid 又把系统属性也纳入了可识别资源的范围之内。

## 应用的上下文

/seapp\_contexts 文件中记录的是：应用（以 UID 的形式）到域（domain）的映射关系。这个文件的作用是：根据 UID 和 seinfo 字段 [这个字段是由 package manager（包管理器）根据 package 的签名密钥以及该密钥在 /system/etc/security/mac\_permissions.xml 中的对应记录填写的] 给应用打上标签。你可以用 toolbox 中的 “ps -Z” 命令来查看进程（应用）的上下文（context），见输出结果 8-6。

1 也就是确定一下文中的各种上下文。——译者注



```

shell@htc_m8w1:/ $ ps -Z | grep platform_app
u:r:platform_app:s0 u0_a42 7343 7129 com.android.systemui
u:r:platform_app:s0 smartcard 7717 7129 org.simalliance.openmobileapi.service:remote
u:r:platform_app:s0 u0_a42 30131 7129 com.android.systemui:recentapp
u:r:platform_app:s0 fm_radio 30405 7129 com.htc.fmsservice
#
# Compare with seapp_contexts
#
shell@htc_m8w1:/ $ grep platform_app /seapp_contexts
user=_app seinfo=platform domain=platform_app type=platform_app_data file
user=smartcard seinfo=platform domain=platform_app type=platform_app_data_file # PID 7717
user=felicarwsapp seinfo=platform domain=platform_app type=platform_app_data_file
user=irda seinfo=platform domain=platform_app type=platform_app_data file
user=fm_radio seinfo=platform domain=platform_app type=platform_app_data_file # PID 30405

```

输出结果 8-6 用“ps -Z”命令查看进程的 SELinux 上下文

## 文件的上下文

SELinux 也能将每个文件与一个安全上下文 (security context) 关联起来。/file\_contexts 文件中就记录了所有受保护文件的 context。用 -Z 参数调用 toolbox 中 ls 程序就能显示它们, 如输出结果 8-7 所示。

```

shell@htc_m8w1:/ $ ls -Z /dev | grep video_device
drwxr-xr-x root u:object_r:video_device:s0 video
crw-rw---- system camera u:object_r:video_device:s0 video0
crw-rw---- system camera u:object_r:video_device:s0 video1
crw-rw---- system camera u:object_r:video_device:s0 video2
crw-rw---- system camera u:object_r:video_device:s0 video3
crw-rw---- system camera u:object_r:video_device:s0 video32
crw-rw---- system camera u:object_r:video_device:s0 video33
crw-rw---- system camera u:object_r:video_device:s0 video34
crw-rw---- system camera u:object_r:video_device:s0 video35
crw-rw---- system camera u:object_r:video_device:s0 video38
crw-rw---- system camera u:object_r:video_device:s0 video39
#
# Compare with /file_contexts definitions
#
shell@htc_m8w1:/ $ grep video_device /file_contexts
/dev/nvhdcp1 u:object_r:video_device:s0 # Left over from the external/sepolicy/file_contexts,
/dev/tegra.* u:object_r:video_device:s0 # even though this is not an NVidia device
/dev/video[0-9]* u:object_r:video_device:s0 # note regular expressions gets all the above

```

输出结果 8-7 用“ls -Z”命令查看文件的 SELinux 上下文

## 属性的上下文

我们在第 4 章中已经介绍过: init 的 property\_service 通过一张写死在源码中的 uid 表<sup>1</sup>, 可以对指定进程能否访问特定的命名空间 (namespace) 中的系统属性做出限制。这是一种非常严格的机制, 但如果不同的 Android 发布版本中想要添加新的系统属性或命令空间的话, 却很难

1 即“property\_perms”数组。——译者注

对它进行修改<sup>1</sup>，以适应新的变化。

不过 SELinux 也提供“execution contexts”这一概念，可以很方便地把它扩展到系统属性上。从 JellyBean 版开始，/init 开始改用一个名为 check\_mac\_perms() 的布尔函数对系统属性进行访问控制。这个函数从两个文件 /data/security/property\_contexts（如果存在的话）和 /property\_contexts 中获取系统属性的上下文，见输出结果 8-8。

```
shell@htc_m8wl:/ $ cat /property_contexts
#line 1 "external/sepolicy/property_contexts"
#####
# property service keys
#
#
net.rmnet0          u:object_r:radio_prop:s0
..
..
sys.usb.config      u:object_r
ril.                u:object_r:rild_pro
net.                u:object_r:system_pro
dev.                u:object_r:system_pro
runtime.            u:object_r:system
hw.                 u:object_r:system_prop
sys.                u:object_r:system_pro
sys.powerctl        u:object_r:powe
service.            u:object_r:system
wlan.                u:object_r:system_pr
dhcp.               u:object_r:system_pr
bluetooth.          u:object_r:bluetoo

debug.              u:object_r:shell_p
log.                 u:object_r:shell_pro
service.adb.root    u:object
..
```

输出结果 8-8 SELinux 中系统属性的上下文

如果你回顾一下第 4 章的内容，会发现：系统属性上下文的内容和“property\_perms”数组中的内容是一模一样的。它们的主要区别在于：在一个单独的文件中记录访问控制配置，使我们能够以便捷得多的方式，修改或改变相关属性的访问控制设置，而无须重新编译/init。

## init 和 toolbox 中使用的命令

回顾一下第 4 章中的内容，Android 的/init 进程提供了许多命令，可以在/init.rc 文件中使用这些命令完成对系统的配置。随着 SELinux 的引入，/init 中又新增了一些对 SELinux 的上下文

1 因为尽管我们可以在源码中修改这个“property\_perms”数组的内容，但是一旦编译完成之后，这个数组中的内容就被写死在/init 这个二进制可执行文件中，不能修改了。如果一定要修改它，只有去修改源码，然后重新编译才行。而不像/property\_contexts 文件中的内容，能在无需重新编译的前提下，修改其中的内容，故有此说。——译者注



进行配置的命令。toolbox 中也新增了功能类似的命令，使我们能在 shell 中对 SELinux 进行修改。表 8-7 中列出了相关的命令。

表 8-7 init 和 toolbox 中使用的与 SELinux 相关的命令

init	toolbox	作 用
无	getenforce	查看当前 SELinux 是否被启用了
setcon <i>SEcontext</i>	无	设置(修改)SELinux 的 context. /init 的上下文为 u:r:init:s0
restorecon <i>path</i>	restorecon [-nrV] <i>pathname...</i>	从参数 <i>path</i> 指定的文件中恢复 SELinux 上下文设置
setenforce [ <i>0 1</i> ]	setenforce [Enforcing permissive 1 0]	启用/禁用强制执行 SELinux
setsebool <i>name value</i>	setsebool <i>name value</i>	设置 SELinux 中的布尔值(0 对应 false/off ; 1 对应 true/on)

注意，你也可以用访问/sys/fs/selinux 目录中存放的各个文件的方式，实现上述 SELinux 命令的大部分功能（而且事实上，上述这些 SELinux 命令也是用类似的方式执行操作的）——尽管要访问这个目录中的文件需要同时拥有 root 权限，并处于 unconfined 安全域（domain）中。/init 进程是处于 unconfined 安全域中的，所以它就可以给其他对象重新设置上下文（比如它可以用 seclabel 参数，给其他服务重新设置上下文），并且可以在/init.rc 文件中的 trigger 语句块中加入设置系统属性 selinux.reload\_policy 的值，以重新加载 SELinux 策略配置的语句。禁用 SELinux 有两个方法：可以通过设置/sys/fs/selinux/disable 这个伪文件的值的方式，来完成任务；也可以通过内核命令行参数 selinux=0 来达到目标（尽管有些厂商在编译安装在它们正式发售的移动设备上的 Android 系统时，会把这个命令行参数给去掉）。

## 其他值得注意的特性

Android 中也启用了 Linux 中的其他一些特性，以增强安全性，特别是对于其他一些不安全的默认设置进行加固。本节中将简短地讨论它们。

### AT\_SECURE

Linux 内核的 ELF 加载器使用了一个辅助向量(vector)，用它在加载二进制可执行文件时，提供该可执行文件的元数据。这个向量可以通过/proc 伪文件系统下的/proc/pid/auxv 文件进行访问。在这个 auxv 文件中有一个名为“AT\_SECURE”的记录项，对于被设置了 setuid 或 setgid 的二进制可执行文件，拥有一定权能的程序以及那些运行之后会引起 SELinux 安全域切换(domain traversal)的程序来说，“AT\_SECURE”记录项中记录的是一个非零的值。在这些情况下，Bionic 的 linker (/system/bin/linker) 配置，会令它屏蔽掉那些“不安全的”环境变量（至于哪些是“不安全的”环境变量这个问题，Android 已经把它们以硬编码的形式，在源码文件



bionic/linker/linker\_envion.cpp 中的 `__is_unsafe_environment_variable()` 函数中一一列出了)。这些环境变量中最主要的两个是：`LD_LIBRARY_PATH` 和 `LD_PRELOAD`——这两个环境变量经常被黑客利用，进行库注入攻击（library injection）。

## 地址空间布局随机化（ASLR）

代码注入攻击是把目标进程的内存地址空间作为施展身手的舞台的。要成功地完成攻击，就必须对内存地址详细布局（内存地址，各种内存段，内存中受保护的区域的出现位置）十分了解。代码注入攻击既可以是向一个已有的进程中加入新的代码，也可以通过跳转到（目标进程中）已存在区域的方式改变目标进程的执行流。不论是在上述哪种情况中，详细了解（目标进程内存空间的）布局都是必要的前提条件——因为如果跳转到了一个错误的地址上，会导致目标进程崩溃。通常进程都是运行在一个地址固定的私有内存段中的，所以黑客可以（套用一句 Java 的广告语）“一次调试，到处黑人”。

地址空间布局随机化（ASLR）就是要通过随机分配各个库（可执行文件）的加载基地址（也就是破坏内存段的地址布局）的方式，增加代码注入攻击的难度。这会让目标代码片段在内存中“到处移动”，这样基本上就能把程序被恶意攻击者控制，并能让攻击者在目标系统上运行任意代码，变为只是让程序崩溃而已——两害相权取其轻。

Linux 系统通过 `/proc/sys/kernel/randomize_va_space`（或者通过 `sysctl.kernel.randomize_va_space`）这一接口，向用户提供了随机化进程地址空间布局的能力。这个值如果是 0 的话，表示不对进程地址空间布局做随机化处理；如果是 1 的话，表示只对栈所在的内存地址空间做随机化处理；如果是 2（这个值是默认值）的话，表示会对栈和堆所在的地址空间做随机化处理。可执行文件也可以指定用 PIE（Position-Independent-Executable，代码的执行与加载地址无关）方式编译（使用 `-pie` 编译选项），从 Android Lollipops 版起，已经（通过在 `Android.mk` 文件中定义 `APP_PIE` 宏的方式）强制要求 Android 中的可执行文件以这一方式编译了。

## 实验：测试 ASLR 有无启用

你只需在 `/proc` 目录中运行下面这个 shell 脚本，就能看到启用 ASLR 后的效果。这个脚本的作用是枚举所有的进程，并把进程中 `libc.so` 这个库的加载基地址（只显示 `.text` 段的起始地址，具体做法是：用 `grep r-x` 执行过滤）和进程的 PID 一起列出，显示在屏幕上，见输出结果 8-9。

如输出结果 8-9 所示，同一个库在不同进程中的加载基地址都是不同的，而且明显是随机分配的。但是我们注意到，仍有一些进程，在这些进程中 `libc.so` 这个库的加载基地址是一样的——出现这一情况的原因是：这些进程都是从 `zygote` 进程中 `fork` 出来的。由于它们是 `fork()` 之后再加载不同的类，才形成不同的进程的，而不是由父进程调用 `exec()` 而产生的，所以它们的

内存地址空间布局还是相同的。从这一点上说, Android 中的 ASLR 比 Linux 系统中的要稍弱些。

```
root@htc_m8wl:~# cd /proc
root@htc_m8wl:/proc # for x in $(cat /dev/urandom | fold -w 1024 | tr -dc 'a-f0-9' | fold -w 1024 | xargs echo | tr -d '\n' | sort); do
    lc=$(grep libc.so /proc/$x/maps | grep r-x | \
    if [[ ! -z "$lc" ]]; then echo $lc in PID $x; fi; \
done | sort
400c6000-40111000 r-xp 00000000 b3:2e 1492 /system/lib/libc.so in PID 28686
.. # All Android apps share shared memory space
400f7000-40142000 r-xp 00000000 b3:2e 1492 /system/lib/libc.so in PID 9615
b6de0000-b6e2b000 r-xp 00000000 b3:2e 1492 /system/lib/libc.so in PID 7470
b6e5a000-b6e55000 r-xp 00000000 b3:2e 1492 /system/lib/libc.so in PID 375
```

输出结果 8-9 显示启用 ASLR 后的效果

大约从 Linux 3.14 版起, 内核级的 ASLR 也已经在 Linux 中被引入了, 但它并没有被多数厂商所采纳。因此在大多数(使用 3.10 或 3.4 内核的) Android 设备中, 并没有提供这一重要的特性, 这也使得 Android 内核中漏洞的利用相当方便。



ASLR 这种防御措施针对的只是以向程序输入一段数据的方式进行代码注入这一攻击手段。如果攻击者已经有了执行代码的权限, 那么他只需调用功能强大的 ptrace(2) API 就能访问到其他进程的内存地址空间, 甚至是远程注入线程。感谢上帝, 要完成这一操作, 攻击者首先必须要有 root 权限。SELinux 可以(而且应该!)用来防止未经授权的用户访问 ptrace(2)。

## 内核加固

与主流 Linux 系统不同, Android 的内核默认不会导出 /proc/kcore——因为用户态进程可以通过这个文件, 以只读的方式访问内核的内存(当然这得有 root 权限)。不过在大多数设备中, /proc/kallsyms 文件还是存在的(而且, 事实上是默认全局可读的)。但它是由系统控制变量 kernel.kptr\_restrict 保护着的, 这个变量的默认值是 2, 这样就能阻止 /proc/kallsyms 文件中的任何数据被显示出来。(用户对)内核中的 ring-buffer 的访问(通过 dmesg)也是以类似的方式被 kernel.dmesg\_restrict 保护起来的。

## 栈保护

虽然攻击可以变得越来越老练, 但是它们的基本攻击形式(在大多数情况下)仍然是: 覆盖掉一个函数指针——这样, 当这个函数被调用时, 就会导致程序的控制流被黑客接管。尽管并不是所有的程序中都会使用函数指针, 但所有的程序中都会使用返回地址(当调用某个函数时, 主调函数会把被调函数返回后要继续执行的下一条指令的所在地址写入栈中), 这就是所谓的“函数返回地址”。

作为一种对栈(stack)溢出攻击的反制手段, 大多数现代编译器都提供一种名为“stack canary”的自动栈保护机制。就像矿工用来探测矿井下是否缺氧的金丝雀(canary)一样, “stack canary”

是写在栈中函数返回值之前的一个随机数，函数能在返回之前检验它是不是被修改过了。如果发现这个“stack canary”变量被修改过了，就说明已经发生了栈溢出，那么程序就会自动终止，而不是冒着可能触发恶意代码的危险继续运行下去。

早在 Android 系统刚出现时，它就已经使用 gcc 的 `-fstack-protector` 编译选项，采取了这种保护措施。不过请注意，这也不是一个一劳永逸的解决方案——因为除了函数返回地址之外，代码注入还能够通过覆盖函数指针的方法实现（C++中的方法<sup>1</sup>就是这类攻击的良好目标）。

## 数据执行保护

在代码注入攻击中，必须要在给程序的输入数据（这个“输入数据”既可以是用户直接输入的数据，也可以是程序打开的其他数据源中的数据）中嵌入恶意的代码<sup>2</sup>。但是，不管怎么说，输入数据总归是数据——而用来存放数据的内存（堆和栈）都可以被标记为“不可执行”（non-executable）。做了这样的标记之后，就能在某种程度上让攻击变得更加困难——因为这时，仅仅使用经典的“蹦床”（trampoline）技术（即，覆盖一个函数指针或是栈中的一个函数返回地址，让它指向黑客注入的代码所在的地址<sup>3</sup>）就会因为注入的代码位于数据段上（因而不可执行）而无法完成攻击。

不幸的是（在大多数情况下），尽管启用数据不可执行保护能够使得一些水平较差的攻击失败，但是攻击手段也是在不断进化的。当前黑客们的反击手段是 ROP（Return-Oriented-Programming）技术。这是一种相当老的技术[最早是由 Alexander Peslyak（Solar Designer）在 1997 年的论文 *Return-into-lib(c) exploits* 中提出的]，是一种把程序中已经存在一个由若干个代码片段“拼接”成的 shellcode，在栈中模拟函数调用的方式依次调用这些代码片段，从而完成攻击的漏洞利用技巧。由于在这一过程中，调用的都是代码，所以也就没有任何东西会被标记为“不可执行”，因而也就能相当可靠地绕过数据执行保护。

## 编译时的防护措施

从某种程度上说，上述的这些保护措施都是针对某种攻击方式，而采用的头疼医头、脚痛医脚式的针对性保护措施。但时至今日，对抗利用内存溢出类漏洞而进行的代码注入攻击的唯一正确方式是：采用防御性编程技术——这涉及进行输入数据验证，并在执行内存操作时实行严格的边界检查。较新版的 Android 系统中已经把它提上了议事日程——在编译源码时加强了

---

1 原文如此，但以译者的观点，这里作者想说的是 C++中的虚方法。——译者注

2 俗称 shellcode。——译者注

3 原文如此，作者这里说得太简单了，即使是最经典的栈溢出，被覆盖的函数返回地址上也是要写一个在内存里找到的“`jmp esp`”指令的地址，然后再靠这条指令跳转到 shellcode 上去的。——译者注



检查，特别是使用 `FORTIFY_SOURCE` 和 `-Wformat-security` 编译开关，增加了对内存拷贝函数的检查，并预防格式化字符串攻击。

## seccomp-bpf

在 Nougat 版中，通过采用 Linux 内核中的 `seccomp-bpf` 机制，引入了一个重要的沙箱特性。在调用了 `prctl(..., PR_SET_SECCOMP)` 之后，`seccomp` 就会被激活，它使用伯克利包过滤（BPF，Berkeley Packet Filter）状态机语法，过滤掉了所有的系统调用（只有那些被认为是“安全的”系统调用才能例外），这就有效地对该进程进行了安全隔离。尽管把进程可以使用的系统调用限制在一定的范围之内，无疑会更安全些，但我们仍须记住，哪怕是 `futex(2)` 这种看似人畜无害的系统调用，在历史上也曾经引发过安全漏洞。

## 8.3 Dalvik 层上的安全措施

### Dalvik 层上的权限

由于运行的是虚拟机代码而不是原生代码，这就给 Dalvik 虚拟机在监视操作和强制安全检查方面带来了极大的优势。如果运行的是原生代码，我们虽然也能靠监视系统调用的方式来获知（程序对）资源的访问情况，但监视系统调用的问题是：它们所能提供的监视粒度不够精确。文件访问（也就只有“打开/读/写/关闭”这几个操作）还算简单，但其他一些操作（比如，完成一次 DNS 查询），由于需要调用多个系统调用，监视起来就费老鼻子劲了。这时，虚拟机的优势就显现无疑了——大多数的操作都是由预先提供的包或类来实现的，而这些包或类中已经自带了权限检查功能。

Android 甚至走得更远：考虑到恶意的开发者可以在一个正常的 Java 类中，显式地导入其他类，并在这些被导入的类中实现一些（恶意的）功能——使用另起炉灶新写的代码，或直接使用 JNI（以逃出虚拟机的监控范围），从而避开权限检查。但是这一做法在 Android 几乎是没什么用的，因为用户应用是没有任何权限的——在 Linux 层上没有任何权限或权能，它对任何系统底层资源的访问都会被直接拒绝。用户应用想要做任何影响会超出应用自身范围的操作时，都必须调用 `getSystemService()` 函数，通过 `system_server` 中的服务完成相关操作。

任何应用都能任意发起调用 `system_server` 中相关系统服务的请求，但是请求是不是会被允许，就不是应用说了算的了——`system_server` 将会对请求是否符合权限规定进行审核。这一审核是在应用自身进程之外进行的，所以除了事先就已经被分配到了的权限之外，应用是没有任何途径能获取相关服务的访问权限的。而权限的分配则是在安装或加载应用时执行的——这也

就意味着用户会看到系统自动弹出一个对话框，提示该应用需要什么权限（谷歌居然神奇地认为用户会把这一长串权限申请清单读完），再由用户同意授予应用这些的权限（当然，谷歌也居然再次神奇地认为用户能够充分了解，点下“确定”按钮会带来什么样的后果）<sup>1</sup>。如果应用运行时所请求的操作因没有相关权限而被拒绝（比如被 AppOps 服务或 pm 拒绝），那么就会抛出一个安全异常（正常情况下，这会导致应用崩溃——除非开发者专门为这类异常编写了异常处理代码，这种情况下，应用将会自己来处理这个异常，通常会弹出一个对话框解释应用需要什么权限，有时也会静静地做好收尾工作，并结束程序）。

接下来要说的是：权限本身是不需要任何特殊的数据结构或复杂的元数据来表示的。在 Dalvik 虚拟机中的权限只不过是一个简单的常量值罢了——这个值是根据应用在它的（AndroidManifest.xml 文件中的）<user-permission>元素中声明的权限，分配在它的 manifest 中的。应用也可以在 Manifest 中的<permission>标签里，定义它自己的常量。当包管理器安装一个应用时，它会把应用自己声明的权限，添加到“权限数据库”中去——这个所谓的“权限数据库”实际上只是包数据库（/data/system/packages.xml）的一部分。这个数据库中除了权限之外，还含有大量（包括各个应用的公钥在内的）极具价值的信息（这也就是为什么我们会在第 2 本中详细讨论它的缘故）。我把其中与权限相关的部分，列在了表 8-8 中。

表 8-8 包数据库中权限相关的元素

元 素	元素中记录的内容
permission-trees	一个树（tree）item 的数组，用来指定权限的命名空间（namespaces），以及定义它们的包
permissions	<p>一个权限 item 数组，其中每个 item 中都有如下几个字段：</p> <ul style="list-style-type: none"><li>• Name——权限常量的名称，也就是它在.apk 文件的 AndroidManifest.xml 文件里声明该权限时使用的名称</li><li>• package——定义该权限的包的名称（如果是 SDK 中提供的权限，那么这个字段就应该填上“android”）</li><li>• protect——这个字段中记录的是一个权限保护的级别，再异或上 PermissionInfo 类中的标志位。权限保护的级别可以是：0 (PROTECTION_NORMAL)、1 (.._DANGEROUS)、2 (.._SIGNATURE) 或 3 (.._SIGNATURE_or_SYSTEM)；PermissionInfo 类中的标志位可以是 SYSTEM(0x10)和 DEVELOPMENT(0x20)。请注意：虽然这个值在这里是以一个十进制数的形式表示的，但它实际上应该是一个十六进制数</li></ul>

---

<sup>1</sup> Android 从 M 版起，终于修正了这个长期以来的错误模型，转而采用 iOS 的设计，在运行时检查权限，并在相关操作发生时，才使用应用自身进程以外的另一个进程来提示用户（在 iOS 系统中，这一操作是由 TCC 守护进程执行的）。——原注

续表

元 素	元素中记录的内容
package	设备中每个已经安装了的应用，在这里都会有一个对应的元素，应用的名称（逆 DNS 形式）记录在 name 字段中，系统分配给该应用的 AID 记录在 userId 字段中。特别要注意的是：赋予该应用的所有权限，全部以 item 的形式，列在了<perms>子元素中
shared-user	如果两个或两个以上的应用共享了某个 AID，那么就会出现一个对应的 shared-user 元素，在该元素中的 userId 属性里记录的就是这个 AID。而且如果这个 AID 被赋予了各类权限，那么这些权限都会以 item 的形式出现在这个元素的<perms>子元素中（赋予一个 AID 权限，实际上就是将该权限赋予了所有共享了这个 AID 的应用）

如果你查看一下包数据库的内容（需有 root 权限），你将会发现在<permission>元素中，既含有定制的权限（即，那些由已经安装了的应用声明的权限），也有系统权限。系统内置的权限（和它们的 protectionLevel 一起），是在/system/framework/framework-res.apk 中指定的，你可以用 appt 工具查看其中的内容，如输出结果 8-10 所示。

```
morpheus@Forge (~/.tmp) % adb pull /system/framework/framework-res.apk
6343 KB/s (19250841 bytes in 2.963s)
morpheus@Forge (~/.tmp) % aapt d xmltree framework-res.apk AndroidManifest.xml | more
N: android-http://schemas.android.com/apk/res/android
E: manifest (line=20)
A: android:sharedUserId(0x0101000b)="android.uid.system" (Raw: "android.uid.system")
A: android:versionCode(0x0101021b)=(type 0x10)0x15
A: android:versionName(0x0101021c)="5.0-1573874" (Raw: "5.0-1573874")
A: android:sharedUserLabel(0x01010261)=@0x1040104 # link to resources.arsc
A: package="android" (Raw: "android")
A: coreApp=(type 0x12)0xffffffff (Raw: "true")
E: uses-sdk (line=0)
A: android:minSdkVersion(0x0101020c)=(type 0x10)0x15 # 21, For Android L
A: android:targetSdkVersion(0x01010270)=(type 0x10)0x15 # 21, For Android L
E: eat-comment (line=27)
E: protected-broadcast (line=29)
A: android:name(0x01010003)="android.intent.action.SCREEN_OFF"
E: protected-broadcast (line=30)
A: android:name(0x01010003)="android.intent.action.SCREEN_ON"
...
# Permission groups, as returned by pm list permission-groups
E: permission-group (line=315)
A: android:label(0x01010001)=@0x1040109 # For UI or pm .. -s
A: android:icon(0x01010002)=@0x1080532 # For UI display
A: android:name(0x01010003)="android.permission-group.MESSAGES"
A: android:priority(0x0101001c)=(type 0x10)0x168
A: android:description(0x01010020)=@0x104010a # for pm list permissions -s
A: android:permissionGroupFlags(0x010103c5)=(type 0x11)0x1 # FLAG_PERSONAL_INFO
E: permission (line=323)
A: android:label(0x01010001)=@0x1040161
A: android:name(0x01010003)="android.permission.SEND_SMS" # For UI or pm .. -s
A: android:protectionLevel(0x01010009)=(type 0x11)0x1 # NORMAL (0), DANGEROUS(1), etc
A: android:permissionGroup(0x0101000a)="android.permission-group.MESSAGES"
A: android:description(0x01010020)=@0x1040162 # for pm list permissions -s
A: android:permissionFlags(0x010103c7)=(type 0x11)0x1 # FLAG_COSTS_MONEY
...
```

输出结果 8-10 从一台 Nexus 9 手机里 dump 出来的/system/framework/framework-res.apk 文件



如输出结果 8-10 所示, 权限是和组以及一对标志位 (FLAG)<sup>1</sup> [这对标志位分别被定义在组上 (group level) (通过 `android.content.pm.PermissionGroupInfo`) 和应用自己身上 (individual level) (通过 `android.content.pm.PermissionInfo`)] 绑定在一起的。对于能够使用 `pm` 脚本 (upcall script) 显示和管理权限的高级用户来说, 这种绑定和分类权限的方式提供了不小的便利性。

## 实验: 使用 `pm` 命令

你可以使用 `pm list permissions` 命令将 (Android 框架以及第三方应用的) 权限输出至屏幕。如输出结果 8-11 所示。

```
root@htc_m8wl:/# pm list permissions -f | more
All Permissions:
+ permission:android.permission.GET_TOP_ACTIVITY_INFO
  package:android
  label:get current app info
  description:Allows the holder to retrieve private information about the current application
               in the foreground of the screen.
  protectionLevel:signature
.. # Application declared permissions, as imported from their AndroidManifest.xml
+ permission:com.facebook.system.permission.READ_NOTIFICATIONS
  package:android
  label:null
  description:null
  protectionLevel:signature
..
```

输出结果 8-11 用 `pm` 命令列出权限

除了 `list` 参数之外, 其他一些有用的参数还有: `-s` (在你的本地列出可供人类阅读的详细信息) 和 `-g` (权限组) 等。`pm` 命令还可以用来执行授权或撤销授权操作 (`pm [grant|revoke] PACKAGE PERMISSIONS`), 甚至可以用来强制启用权限 (`pm set-permission-enforce PERMISSION [true|false]`)。我们在第 2 本中将详解这条命令的语法以及在 Android M 版中引入的一些值得注意的改变。

AppOps 服务 (将在第 2 本中详细讨论它) 提供了一个强大的 GUI 界面。通过它, 用户可以追踪和 (在相当细的粒度上) 调整应用权限的使用。作为 KitKat 4.4.2 “安全升级” 的一部分, 这个 GUI 已经被移除了, 但这个服务仍然还在并且工作良好。事实上, Lollipop 版中引入了一个名为 AppOps 的脚本 (upcall script), 这个脚本可以用来允许、拒绝、忽略或重置应用的权限。可惜的是, 这个命令执行程序只支持一小部分操作 (`android:[coarse|fine|monitor]_location`、`android:get_usage_stats` 和 `android:activate_vpn`), 不过只需重新编译一下

1 虽然名称是“标志位 (FLAG)”, 但实际上, 当前这对“标志位”的取值是唯一的: 在 `permissionGroupFlags` 中总是 `FLAG_PERSONAL_INFO`, 在 `permissionGroupFlags` 中总是 `FLAG_COSTS_MONEY`。不过这一设计, 留下了将来对它进行扩展的余地。——译者注

android.app.AppOpsManager 就可以让它支持所有的（当前一共 48 个）操作了。不过请注意：AppOps 是权限栈顶端的另外一层，它使用的是一个独立的数据库（/data/system/appops.xml）。如输出结果 8-12 所示。

```
shell@flounder:/ $ appops
usage: adb shell appops set <PACKAGE> <OP> <allow|ignore|deny|default> [--user <USER_ID>]
<PACKAGE> an Android package name.
<OP> an AppOps operation.
<USER_ID> the user id under which the package is installed. If --user is not
specified, the current user is assumed.
shell@flounder:/ $ appops set com.android.musicfx android:get_usage_stats allow
# To see changes reflected in the AppOps database, you need root:
shell@flounder:/ $ su
root@flounder:/ # cat /data/system/appops.xml | grep -A 4 musicfx
<pkg n="com.android.musicfx">
  <uid n="10014" p="true">
    <op n="0" />
    <op n="43" m="0" /> # android.apps.AppOpsManager.OP_GET_USAGE_STATS = 43
  </uid>
```

输出结果 8-12 在 Android Lollipop 版中使用 AppOps 脚本的例子

## 把 Dalvik 虚拟机层上的权限映射到 Linux UID 上

Dalvik 层面上的权限到 Linux 层面上的权限之间的“桥梁”是/system/etc/permissions/platform.xml 文件。这是个包含在 AOSP 源码中的文件，并有良好的文档支持，以便厂商能够（谨慎地）往其中添加特定的权限或 AID。映射可以以两种方式进行：可以用<permission>标签，把 Dalvik 上使用的高级权限<sup>1</sup>与记录在<group>标签中的 gid 关联起来；也可以用<assign-permission>把某个高级权限赋予某个 uid。代码清单 8-4 中显示的就是该文件的一个样例。

如果你查看你自己的移动设备上的/system/etc/permissions/目录，或许会看到目录中有另外几个 XML 文件——android.hardware.\*和 android.software.\*，这些文件都是在 build 系统的过程中，从 AOSP 源码文件中复制过来的，当然其中有些也可能是厂商加进来的文件。

<sup>1</sup> Dalvik 中的这个“高级权限”中的“高级”指的是：在“高级”语言也就是 Java 语言中使用的权限。——译者注

代码清单 8-4 /system/etc/permissions/platform.xml 文件的一个例子

```

...

<!-- This file is used to define the mappings between lower-level system
user and group IDs and the higher-level permission names managed
by the platform.

    Be VERY careful when editing this file! Mistakes made here can open
    big security holes.
-->
<permissions>

    <!-- The following tags are associating low-level group IDs with
    permission names. By specifying such a mapping, you are saying
    that any application process granted the given permission will
    also be running with the given group ID attached to its process,
    so it can perform any filesystem (read, write, execute) operations
    allowed for that group. -->

    <permission name="android.permission.BLUETOOTH_ADMIN">
        <group gid="net_bt_admin" />
    </permission>

    <!-- ===== -->

    <!-- The following tags are assigning high-level permissions to specific
    user IDs. These are used to allow specific core system users to
    perform the given operations with the higher-level framework. For
    example, we give a wide variety of permissions to the shell user
    since that is the user the adb shell runs under and developers and
    others should have a fairly open environment in which to
    interact with the system. -->

    <assign-permission name="android.permission.MODIFY_AUDIO_SETTINGS" uid="media"/>
    ...

    <!-- This is a list of all the libraries available for application
    code to link against. -->

    <library name="android.test.runner"
        file="/system/framework/android.test.runner.jar" />

</permissions>

```

## Dalvik 代码签名

光有权限本身，用处还不是最大的——毕竟，所有的应用都能在它的 `AndroidManifest.xml` 中声明它需要使用哪些权限，而一些不怎么靠谱的用户也可能在提示框弹出后，看也不看就直接去点“确定”按钮。为了保证安全性，谷歌规定所有上传到谷歌 Play 商店中的应用都必须经过数字签名，这样就能验证应用开发者的身份，并在必要时能够追究相关的责任。

因此，所有的 Android 应用都必须经过签名（签名的具体操作过程详见第 2 本）。不过签名的人到底是谁？这个问题却不是很明确。谷歌出于同苹果竞争的考虑，开放了谷歌 Play 商店，希望通过简化发布验证流程的形式，吸引开发者。相对于苹果公司冗长的发布验证流程（所有的应用都必须通过苹果公司的审查，并对它们做数字签名），谷歌则开放得多，它让任何人都能对应用进行数字签名），只要创建一对（公-私）密钥，公开他们的公钥，并用私钥对他们的 APK



文件做数字签名就行了。这样做的理由是：这种做法能在仍然保证可以有效验证 APK 的作者的前提下，极大的简化将应用提交到商店中的流程。

但在实际操作中，这一做法使得谷歌 Play 商店里恶意软件的数量出现了爆炸式的增长。谷歌处理这一问题的对策是：一旦发现或有人举报恶意软件，就直接把它从商店里删掉，同时把对应的公钥加入黑名单里去。但要是站在恶意软件作者的角度，“坏事就是要先做了再说”，因为恶意软件从上架到被发现，中间有一个过程，这段时间很可能已经让恶意软件传播出去了，恶意软件作者的目的就已经达到了。再加上恶意软件的开发者总是能再次生成新的密钥对这一事实，就直接掏空了整个安全模型。最近（发表在 RSA 2014 会议上<sup>[4]</sup>）的研究结果表明：“从 2011 年到 2013 年，恶意应用的数量增长了 388%，而谷歌每年删除的恶意应用的数量，则从 2011 年的 60% 下滑到了 2013 年 23%”。实际上，在谷歌的 Play 商店里，大约每 8 个应用中就有一个是恶意应用。

## Android 的 Master Key 漏洞

2013 年间，发现了 Android 历史上最严重的漏洞中的一个，也就是后来（多少有些错误地）被称为 Master Key 漏洞。引发这个漏洞[由 BlueBox security<sup>[5b]</sup>发现，并由大名鼎鼎的 iOS Cydia 软件的作者 Saurik<sup>[5b]</sup>（及其他人）提出了更巧妙的利用方法]的 bug 是：如果 APK 文件中存在多个同名文件，Android 是不能正确处理这种 APK 文件的。APK 文件实质上是一个 ZIP 文件，而包括 aapt 在内的大多数程序是不允许在一个 ZIP 压缩包中存在两个文件名相同的文件的。但是从技术上说，这种情况是有可能发生的——而且一旦发生，就会引发一个怪异的漏洞：文件签名的验证是 APK 文件中的一个文件，但是提取和安装的却是 APK 文件中的另一个同名文件。出现这个奇怪的漏洞的根本原因是：Android 中是用两个不同的库（Java 的和 Dalvik 的原生实现代码）来分别完成验证文件签名和从 APK 文件中提取文件这两个任务的。结果，这就让任何人都能在拿到一个已经经过签名的 APK 文件后，往里面添加一些与 APK 文件中原有的文件同名的文件（其中当然也包括 classes.dex）。如此一来，就能有效地绕过 Android 对 APK 的签名验证机制。尽管这个漏洞现在已经被修复了，但它却是一个很好的案例：系统中会时不时地被曝出一些安全漏洞出来，而要利用这些漏洞去做些坏事，却几乎不需要什么技术储备。

## Android 的 Fake ID 漏洞

2014 年，与 Master Key 漏洞对应的当数 Fake ID 漏洞了。这一次引发问题的是 Android 在验证 App 的签名证书中的一个缺陷。通过故意提供一个不正确的证书链，使得攻击者能够伪造 App 的签名证书：把一个恶意应用（用伪造的证书签名）和一个（甚至是多个）真实的（但和恶意应用完全无关的）应用关联在一起。结果，恶意应用就能继承到原本分配给可信应用的权限（在给出的例子中，恶意应用通常会把自己伪装成 Adobe 的组件和 WebKit 的插件）。

这个漏洞（它也是 BlueBox<sup>[6]</sup>发现的）在当年的 BlackHat 大会上引起了一场地震，尤其是考虑到：它已经存在了近 4 年的时间（从 Eclair 开始，一直到当年的最新版 Lollipop），当时市场中所有的 Android 设备都受到这个漏洞的影响。尽管谷歌最终修补了这个问题，使得 Lollipop 版不再受到这个漏洞的困扰，但是它（和其他许多例子一起）仍然显示了安全漏洞是多么的泛滥成灾！<sup>1</sup>

## 8.4 用户层上的安全措施

本章讨论至此，都着眼于应用层上的安全措施。但 Android 同样也需要在用户层上提供安全保障——只允许合法的设备用户才能访问设备，特别是存储在设备中的敏感数据。从 JellyBean 版开始，Android 开始支持多用户了，这就使情况变得更复杂了些。

### 锁屏机制

对于小偷或者其他能够在物理上接触到移动设备的恶意攻击者来说，锁屏保护是设备的第一道，也是唯一一道真正的防线。同时，在用户唤醒设备时（这是一个非常常用的操作），通常也需要接触到屏幕。因此，这一方案既要有足够的弹性，又要能够让用户自然快速地完成解锁操作。和大多数 Android 特性一样，厂商也可以定制解锁机制——尽管 Android 已经提供了一个屏幕锁的实现，在这一实现中，经常使用的解锁方式有如下几种。

#### 口令、PIN 码和图形锁

默认的 Android 屏幕解锁方案，可以是口令、PIN 码或图形锁。图形锁实际上也是一种 PIN 码，只不过是在使用图形锁时，用户不需要再记忆一串数字，而是只需（通常是在一个 3×3 的点阵上）画出一个图形来就可以了。用户也可以用 PIN 码来代替图形锁——从技术上说，PIN 的安全性比图形锁的更高些。因为 PIN 码最多可以由 16 个数字组成，而且其中可以有重复的数字。而口令的安全性又超过了 PIN 码——因为在口令中可以使用同时使用了大小写和数字的口令。

锁屏界面实际上只是一个实现在 `com.android.keyguard` 包中的 `activity`。在这个包中含有所有系统支持的锁屏方案和方法，包括表 8-9 所列的这些类。

---

1 无独有偶，苹果公司的 iOS 6.x-7.0.4 也受到一个类似的、令人尴尬的 bug 的困扰——这个 bug 被称为 SSL "goto fail"，它是因为在代码中不小心多写了一句 `goto fail` 语句，而使得 SLL 证书验证会被有效地绕过的缺陷。苹果因此也被 Andro-philes 嘲笑：“……说明，住在玻璃房子里的人肯定不会扔石子。”——原注

表 8-9 com.android.keyguard 中的类

类 名	作 用
BiometricSensorUnlock	供生物识别（比如，刷脸解锁）方案使用的接口
Keyguard [PIN SimPin Password] View	要求用户输入 PIN 码或口令的默认 view
KeyguardSecurityView	各种 Keyguard view 的实现代码（在 view 中模拟了 activity 的整个生命周期）
KeyguardService	Keyguard 服务的实现代码
KeyguardSecurityCallback	KeyguardHostView 实现的接口
KeyguardViewMediator	把 Mediates 发送给 Keyguard view

在电源管理模块唤醒屏幕并通知 WindowPolicyManager 实现代码的第一时间里，屏幕解锁代码就会被调用。这会调用 KeyguardServiceDelegate 的 onScreenTurnedOn()方法，接着这个方法又会去调用 keyGuard，并等待它执行完毕并返回。从这里开始，控制权就交给了 keyGuard，它会去确定用户选择的是哪种屏幕解锁方案，并画出相应的屏幕解锁界面（通过相应的 activity）。当系统策略要求强制自动锁屏时，锁屏代码也可以被 DevicePolicyManager 中的 lockNow()方法调用。

实际的解锁操作是由 LockPatternUtils 来完成的，它会请求使用 LockSettingsService 服务（这个服务是 system\_server 中的一个线程），该服务将会验证用户的输入与 LOCK\_PATTERN\_FILE（这个宏指的是 gesture.key 文件，用户使用图形锁时，图形锁的 Hash 存放在该文件中）或 LOCK\_PASSWORD\_FILE（这个宏指的是 password.key 文件，用户使用 PIN 码或口令解锁时，相应的 Hash 存放在该文件中）文件中的记录是否一致，以判定用户的输入是否正确。不论是在哪种情况下，文件中都不会记录图形锁或口令的明文，而是只记录它们的 Hash。另外 LockSettingsService 服务还需使用 locksettings.db 文件，该文件是一个 SQLite 数据库文件，其中记录了与屏幕解锁相关的各种设置，如表 8-10 所示。

表 8-10 locksettings.db 数据库中的键

LockPatternUtils 中的常量	键名 (lockscreen.*)
LOCKOUT_PERMANENT_KEY	lockedoutpermanently
LOCKOUT_ATTEMPT_DEADLINE	lockedoutattempteddeadline
PATTERN_EVER_CHOSEN_KEY	patterneverchosen
PASSWORD_TYPE_KEY	password_type



续表

LockPatternUtils 中的常量	键名 (lockscreen.*)
PASSWORD_TYPE_ALTERNATE_KEY	password_type_alternate
LOCK_PASSWORD_SALT_KEY	password_salt
DISABLE_LOCKSCREEN_KEY	disabled
LOCKSCREEN_BIOMETRIC_WEAK_FALLBACK	biometric_weak_fallback
BIOMETRIC_WEAK_EVER_CHOSEN_KEY	biometricweakeverchosen
LOCKSCREEN_POWER_BUTTON_INSTANTLY_LOCKS	power_button_instantly_locks
LOCKSCREEN_WIDGETS_ENABLED	widgets_enabled
PASSWORD_HISTORY_KEY	passwordhistory

综上所述，图 8-4 中给出的就是一张稍经简化的设备解锁流程图。

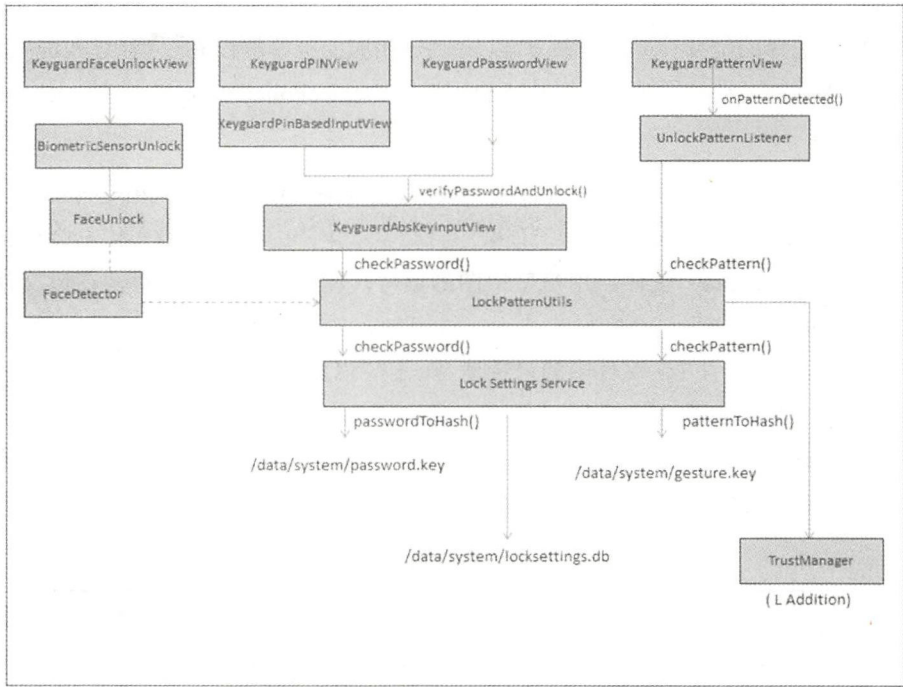


图 8-4 设备解锁流程图

（图 8-4 中右下角的）TrustManager 是 Android L 版中新增的一个功能，它让用户能够以其他非口令/PIN 码/图形锁的方式（比如用一个之前配对好的蓝牙狗或其他蓝牙设备）解锁移动设备。

## 实验：观察 locksettings.db 数据库

如果你的设备已经 root 了，而且设备上还安装了 SQLite3 这个二进制可执行程序，那么你就可以直接在设备上查看 locksettings.db 文件中的内容了。当然，你也可以用 adb 把这个数据库文件 pull 到你的电脑上，然后在电脑上查看其中的内容，见输出结果 8-13。

```
root@htc_m8wl:/data # sqlite3 /data/system/locksettings.db
SQLite version 3.7.11 2012-03-20 11:35:50
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .dump
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE android_metadata (locale TEXT);
INSERT INTO "android_metadata" VALUES('en_US');
CREATE TABLE locksettings (_id INTEGER PRIMARY KEY AUTOINCREMENT,
                           name TEXT,user INTEGER,value TEXT);
INSERT INTO locksettings VALUES(2,'lockscreen.options',0,'enable_facelock');
INSERT INTO locksettings VALUES(3,'migrated',0,'true');
INSERT INTO locksettings VALUES(4,'lock_screen_owner_info_enabled',0,'0');
INSERT INTO locksettings VALUES(5,'migrated_user_specific',0,'true');
INSERT INTO locksettings VALUES(9,'lockscreen.patterneverchosen',0,'1');
INSERT INTO locksettings VALUES(11,'lock_pattern_visible_pattern',0,'1');
INSERT INTO locksettings VALUES(12,'lockscreen.password_salt',0,'-3846188034160474427');
INSERT INTO locksettings VALUES(81,'lockscreen.disabled',0,'1');           # No Lock
INSERT INTO locksettings VALUES(82,'lock_fingerprint_autolock',0,'0');
INSERT INTO locksettings VALUES(83,'lockscreen.alternate_method',0,'0');
INSERT INTO locksettings VALUES(84,'lock_pattern_autolock',0,'0');
INSERT INTO locksettings VALUES(86,'lockscreen.password_type_alternate',0,'0');
INSERT INTO locksettings VALUES(87,'lockscreen.password_type',0,'131072'); # PIN
INSERT INTO locksettings VALUES(88,'lockscreen.passwordhistory',0,'');
DELETE FROM sqlite_sequence;
INSERT INTO "sqlite_sequence" VALUES('locksettings',88);
COMMIT;
```

输出结果 8-13 查看锁屏数据库中相关的设置

locksettings 表中包含一个“user”字段（这个字段是 JellyBean 版中引入的，引入的目的是为了支持 Android 的多用户登录），这个字段的值通常不是 0，就是 1，但有时（在系统中有多用户时）它记录的也会是大于 1 的值——数据库中还有一些组合标志位以及 key 文件的 salt。尽管锁屏设置服务会把这些设置缓存一份，但你仍然可以直接在 SQLite3 中，用 SQL 语句修改锁屏设置。你也可以更改这个文件的文件名——如果你真这么做了，并在此之后重启了 system\_server，那么系统就会用默认值，重新创建一个 locksettings.db 数据库（这就有个很好的副作用——你的锁屏密码或图形锁必须要重新设置了）。

## 其他锁屏方案

Android 的冰激凌三明治（Ice Cream Sandwich）版中引入了人脸识别身份认证技术，作为对传统锁屏方式的补充。这一技术，作为 Android 有，而 iOS 没有的技术之一，被吹捧得太过了些。不幸的是，识别的成功率远远达不到完美的程度——所有的数据都徘徊在 60%到 90%这

样一个较低的水平上。要对付人脸识别技术其实非常简单，只要让手机摄像头对准一张照片就行了。而且有意思的是，尝试过这一做法的人们发现：对着照片识别时的识别成功率竟然比对着一张真正的人脸识别时的成功率还要高！

摩托罗拉的 Atrix 4G 手机是第一款实现了指纹扫描身份认证功能的 Android 设备——这一功能也可以替代传统的锁屏方案。这一技术也饱受较低的正确识别率之苦。苹果公司于 2012 年收购了 AuthenTec 之后暗示，在 iOS 上将会使用指纹识别技术，而且它也确实出现在了 iPhone 5S 上。一开始，三星嘲讽说这个特性太低级，太没有新意，但（毫不令人意外地）它嘴上虽然这么说着，实际上却马上在它的“下一代旗舰版手机”——Galaxy S5 中引入了这一功能。其他的 Android 设备厂商也迅速跟进。随着 Android L 版中内置了一个支持指纹扫描的服务——fingerprint，指纹扫描似乎已经成了 Android 的一种标配特性了。

L 版中新增的另一个重要的特性是：可以用另一个设备 [比如 Android Wear 之类的蓝牙设备（必须事先配对好）] 解锁移动设备。这当然对工作距离有一定的限制——只有当用户出现在设备附近时，设备才会被解锁。TrustManager、fingerprint 以及系统内部的 LockSettings 我们都将在第 2 本中予以详细讨论。

## 支持多用户

在出现以后的很长一段时间里，Android 都是在“一台设备只有一个用户”的前提假设下进行操作的。与支持多个用户账户登录及用户账户间切换已经很长时间的台式机操作系统不同，Android 一直到 JellyBean (4.2) 版才开始引入对多用户的支持，而且还只有在平板电脑上才有这一功能。

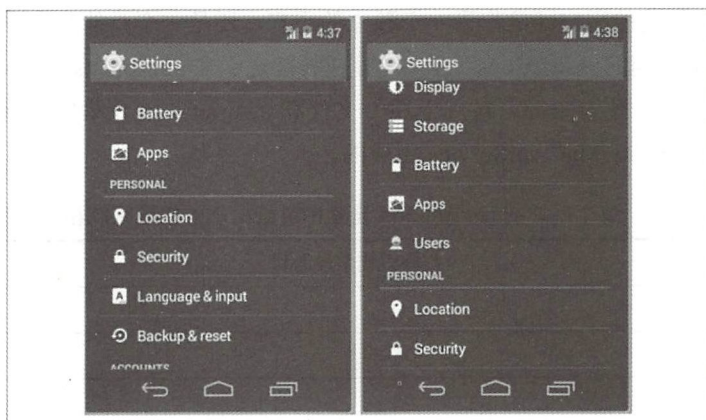
如前文所述，Android 已经把用户 ID 用在各个应用身上了。为了实现同样是构建在用户 ID 上的对多用户的支持，（并使之与前者不冲突）Android 将 AID 空间划分成了几个相互不重叠的区域，并且把其中的一个区域单独拿出来，专供人类用户账户使用。由此，应用的 ID 也从原来的 app\_###改成了现在的 u###\_a###的形式。此外，每创建一个用户账户，都会在/data/user 中创建一个对应的专用账户目录。应用的数据目录也被移到了/data/user/###/，而设备初始化时设置的用户账户就是用户“0”。同时原来存放应用数据的目录/data/data 现在也就成了用户 0 的目录 (/data/user/0 做了一个指向它的符号链接)。用户对自身的配置则被存放在/data/system/users 目录中，如下面这个实验所示。

### 实验：在 Android API 17 及更高版本中启用多用户支持

从 JellyBean 版 (API 17) 开始，在平板电脑中，对多用户的支持是默认启用的。但是大家



相对知之甚少的一个特性是：在手机上，我们也能启用对多用户的支持。我们所要做的只是把一个系统属性 `fw.max_users` 设为比 1 更大的数值就可以了。在 Android 模拟器上执行这一操作，将使 Settings（设置）界面中多出一个 Users（用户）选项出来，如屏幕截图 8-1 所示。



屏幕截图 8-1 设置系统属性 `fw.max_users` 前后的差别对比

添加用户的操作是很方便的，只不过系统会要强制要求你进行锁屏设置——以便在登录时区分到底是哪个用户在登录。设置完毕之后，你可以看到与输出结果 8-14 中类似的信息。

```
root@generic:/data/system/users # ls -F
0/
0.xml
userlist.xml
root@generic:/data/system/users # setprop fw.max_users 3
#
# Add another user through settings.. (shell stop/start might be necessary) then ls again
#
root@generic:/data/system/users # ls -F
0/
0.xml
10/
10.xml
userlist.xml
root@generic:/data/system/users # cat userlist.xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<users nextSerialNumber='11' version='4">
  <user id="0" />
  <user id="10" />
</users>
root@generic:/data/system/users # cat 0.xml # Display details for user 0
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<user id="0" serialNumber="0" flags="19" created="0" lastLoggedIn="1400702272027"
  icon="/data/system/users/0/photo.png">
  <name>Owner.com</name>
  <restrictions /> # restrictions, if any, go in this element
</user>
root@generic:/data/system/users # ls -l 0 # Show settings for user 0
-rw-rw---- system system ... accounts.db # External (POP3, IMAP, etc) accounts
-rw-rw---- system system ... accounts.db-journal # SQLite3 journal
-rw-rw---- system system ... appwidgets.xml # Installed widgets
-rw-rw---- system system ... package-restrictions.xml # Package Restrictions
-rw-rw---- system system ... photo.png # User selected photo
-rwx-rw---- system system ... wallpaper # Selected wallpaper
-rw-rw---- system system ... wallpaper_info.xml # Metadata
```

输出结果 8-14 列出 `/data/system/users` 中存放的多用户配置信息

我们还可以用命令行中的 `pm create-user` 命令添加用户，达到同样的效果。`pm create-user` 命令将会连接用户管理器 [`IUserManager.Stub.asInterface(ServiceManager.getService("user"))`]，并调用其中的 `createUser()` 方法。用与之对应的 `pm remove-user` 命令，则可以删掉一个用户。

根据你是如何创建的用户（是作为一个独立的用户，还是一个只能共享原来用户的 App 的受限用户），表 8-11 所列的 `restrictions` 元素中的这些（被定义在 `android.os.UserManager` 类中的）布尔属性值有可能会被设为 `true`。对（上述）文件和 `restrictions` 元素的实际处理操作，是由 `com.android.server.pm.UserManagerService` 完成的。

表 8-11 每个用户的 `restrictions` 元素中可能出现的权限限制相关字段

权限限制字段名称
<code>no_modify_accounts</code>
<code>no_config_wifi</code>
<code>no_install_apps</code>
<code>no_uninstall_apps</code>
<code>no_share_location</code>
<code>no_install_unknown_source</code>
<code>no_config_bluetooth</code>
<code>no_config_credentials</code>
<code>no_remote_user</code>

## 密钥管理

无论是供系统内部使用（验证当前安装的 `apk` 文件的签名），还是供 App 使用，Android 都极度依赖加密密钥。无论是在上述哪种情况下，`keystore` 服务（详见第 4 章）在抽象和隐藏具体实现方面都扮演着主要的角色。

## 证书管理

公钥基础设施是所有互联网安全事实上的支点。加密依赖于几个关键假设，这些关键假设是：关于算法，算法背后的公钥，以及最重要的可信性的。简而言之，如果你知道某个主体（`subject`）的公钥，那么这个公钥不光可以用来加密要发送给该实体的消息，而且还可以用来验证某个消息是不是来自该实体的（数字签名）。由此可以做出进一步的推论：如果该实体愿意为另一个（实体的）公钥做担保 [通过对该公钥进行数字签名的方式（实际上经过数字签名的这个就是“证书”），那么公钥之间的信任关系就这样建立起来了。用这种方式，信任的层级传递关系就可以

建立起来了。

这一原理尽管十分强大，但却面临一个“先有鸡还是先有蛋”的问题——你可以用一个公钥为其他公钥提供认证（数字签名）服务，但这个公钥却必须事先由其他的公钥对它进行认证（签名）。解决这一窘境的方法是：用硬编码的方式，在操作系统中写死几个最初的公钥。这些公钥是以根证书（root certificate）（也就是能自己认证自己的公钥）的形式编码的。它们不会经网络传递，否则它们就没有价值了（因为这样很容易受到欺骗，被攻击者把真的证书替换成假的证书）。但是如果它们是硬编码写死在操作系统中的，它们就是可信的，并且可以以它们为基础，构造信任传递链。

Android 硬编码写死的根证书是放在 `/system/etc/security/cacerts` 中的，这些证书是以 PEM（Privacy-Enhanced-Mail）（一种以分隔符标识出证书，并以 base64 算法对证书进行编码的格式）的形式存放的。在有些设备上，也会有 PEM 编码之前或之后的 ASCII 明文形式的证书。如果没有的话，用命令行实用程序 `openssl`（Linux 或 Mac OS 操作系统中都会自带这个工具）显示出其中的内容也只是举手之劳，如输出结果 8-15 所示。

```
morpheus@Forge (/tmp)$ adb pull /system/etc/security/cacerts
pull: building file list...
pull: /system/etc/security/cacerts/ff783690.0 -> ./cacerts/ff783690.0
...
morpheus@Forge (/tmp)$ openssl x509 -in ff783690.0 -text | more
Certificate:
    Data:
        Version: 3 (0x2)      # Denotes the X.509v3 format
        Serial Number:       # Used to refer to certificate when revoking
                             44:be:0c:8b:50:00:24:b4:11:d3:36:2a:fe:65:0a:fd
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: # Issuer in LDAP notation: C=country, ST=state, L=location,
               O=Organization, OU= Organizational Unit, CN=Common Name
        Validity
            Not Before: # Usually coincides with certificate issue date
            Not After : # Usually set to 2 10 of years from issue date
        Subject: # Certificate Owner, in same LDAP notation
               # ...
        Subject Public Key Info:
            .. # Modulus and Exponent (usually 65537)
        X509v3 extensions:
            X509v3 Key Usage:
                Digital Signature, Non Repudiation, Certificate Sign, CRL Sign
            X509v3 Basic Constraints: critical
                CA:TRUE
            X509v3 Subject Key Identifier:
                A1:72:59:26:4B:28:98:43:95:5D:07:37:D5:85:96:9D:4B:02:C3:45
            X509v3 CRL Distribution Points:
                URL:http://crl.usertrust.com/UTN-USERSFirst-Hardware.crl
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, IPSec End System, IPSec Tunnel, IPSec User
        Signature Algorithm: sha1WithRSAEncryption
        # ... SHA-1 hash of certificate, signed with RSA private of issuer
    -----BEGIN CERTIFICATE-----
    MIIE4DCCA1ygwIRAgIQRIAMl1AAJLQROzYq/mUK/TANBgkqhkiG9w0BAQUFADCB
    lzEIMakGAlUEBhMCVWxkCzAIBgNVBAAgTAlVUMRcwFQYDVOQHEwEYTVWx0IERxhu2Ug
    ... Base 64 (original PEM) encoding of the certificate
    KgMIDP+JjnlfiyUhlxUdqWqeUQ0qUZ6B+dQ7XnAsfxAynB67nihmqA==
    -----END CERTIFICATE-----
```

输出结果 8-15 使用 openssl 解开一张 PEM 编码后的证书

1 证书前后的分隔符为“-----BEGIN CERTIFICATE-----”和“-----END CERTIFICATE-----”。——译者注



另有一个特别重要的证书是存放在 `/system/etc/security/otacerts.zip` 中的 OTA (Over-The-Air) 升级证书。这个 zip 文档中含有 1 个（在极少数情况下会有多个）用来验证 OTA 升级包的证书（详见第 3 章）。`RecoverySystem` 类会用它的 `getTrustedCerts()` 方法（使用 `CertificateFactory` 参数）解析这个文件（文件的路径已经写死在 `DEFAULT_KEYSTORE` 中了）。再说一遍，所有的证书都是以 PEM 方式编码后存放的（通常其中不会有人类可以直接阅读的文字），但是你可以使用输出结果 8-15 中给出的方法把它解码出来。如果要在某些 Android 系统（比如 FireOS）中禁止系统自动升级（以防止系统升级后，之前对系统做的 root 失效），直接删掉这个文件是个不错的主意。

## 证书 pinning 校验

JellyBean 版（API 17）中引入了证书 pinning 校验（certificate pinning），现在它已经成了 SSL 证书验证的一个常用插件了。证书 pinning 校验就是把目标主机的预期公钥（通过其证书获得）预先写死在系统中，并将其称为证书锁（pin），这样如果该主机出示的证书与证书锁（或者与证书锁集合中的任意一个都）不匹配的话，就会禁止与该主机通信。

与之前讨论的，存放在 `/system/etc/security` 目录（该目录是只读的）中的证书不同的是，证书锁（pin）是把相关公钥记录在 `/data/misc/keychain/pins` 这个可以被修改的文件中的。`CertPinInstallReceiver` 类注册了一个监听 `UPDATE_PINS` 类型的 intent 的 broadcast receiver。在 `UPDATE_PINS` 类型的 intent 中，应有通过调用 `putExtra()` 方法加入的以下这四个内容：

- `EXTRA_CONTENT_PATH`——pin 文件中要新增的“主机-公钥”记录项。
- `EXTRA_VERSION_NUMBER`——这个版本号应该比当前版本号更大<sup>1</sup>。
- `EXTRA_REQUIRED_HASH`——当前 pin 文件的 Hash。
- `EXTRA_SIGNATURE`——新的 pin 文件的数字签名，其中含有它的版本号和当前 pin 文件的 Hash。

`CertPinInstallReceiver` 的 `onReceive()` 方法（继承自 `ConfigUpdateInstallerReceiver`）从以广播的形式发来的 intent 处获取相应的值，并检查 intent 中的版本号是否大于当前 pin 文件的版本号（记录在 `/data/misc/keychain/metadata/version` 文件中），且当前文件的 hash 是否与 intent 中传进来的 Hash 一致。然后，它会使用存储在系统设置数据库中的 `config_updata_certificate`（被定义在变量 `UPDATE_CERTIFICATE_KEY` 中）的证书，去验证签名。如果所有的验证都通过了的话，intent 中 `EXTRA_CONTENT_PATH` 中的内容就会被添加到 pin 文件中，而且 `metadata/version` 中存放的版本号也会更新到 intent 中的 `EXTRA_VERSION_NUMBER` 规定的版本号。

---

1 从本章参考文献[7]中给出的代码看，这个值也可以等于当前版本号。——译者注

谷歌默认会把它自己所有的证书锁（真的有好多）都放进 pin 文件中要求（在使用过程中）执行证书 pinning 校验，厂商也可能往这个文件里再加上些其他的站点的证书锁。输出结果 8-16 中给出的就是快速查看 pin 文件中记录的相关站点域名的方法。

```
root@htc_m8wl:/ # cat /data/misc/keychain/pins | cut -d"=" -f1
*.spreadsheets.google.com
*.chart.apis.google.com
appengine.google.com
*.google-analytics.com
*.doubleclick.net
*.chrome.google.com
*.plus.google.com # largely unused ;- )
# .....
*.youtube.com
*.profiles.google.com
*.mail.google.com
www.googlemail.com
gmail.com
```

输出结果 8-16 显示 pin 文件中使用证书 pinning 校验的站点的域名

在“Android Explorations”博客<sup>[7]</sup>中有一个示例应用，演示了如何创建一个 pin 文件以及如何通过 intent 进行更新操作。

## 证书黑名单

Android 提供了一个 CertBlacklist 类来处理与证书黑名单相关的操作（加入黑名单和移出黑名单）。这个类（是 system\_server 中一个服务的对应框架，详见第 5 章）注册了监视下列这两个 content URI 的 ContentObserver 对象：

- content://settings/secure/pubkey\_blacklist——这个 content 中存储了已知被黑掉的或是已经被吊销了的公钥或证书。这个 content 中的数据实际是存放在 /data/misc/keychain/pubkey\_blacklist.txt 中的。
- content://settings/secure/serial\_blacklist——在这个 content 里记录了已知被黑掉的或是已经被吊销了的公钥的序列号（serial number），这个 content 中的数据实际是存放在 /data/misc/keychain/serial\_blacklist.txt 中的。

上述这两种数据也会被记录在系统的安全设置数据库中，如输出结果 8-17 所示。

```
root@htc_m8wl:/ # sqlite3 /data/data/com.android.providers.settings/databases/settings.db \
"select * from secure" | grep black
95|serial_blacklist|827,864
99|pubkey_blacklist|5f3ab33d55007054bc5e3e5553cd8d8465d77c61,783333c9687df63377efceddd82efa..
root@htc_m8wl:/ # cat /data/misc/keychain/serial_blacklist.txt
827,864
root@htc_m8wl:/ # cat /data/misc/keychain/pubkey_blacklist.txt
5f3ab33d55007054bc5e3e5553cd8d8465d77c61,783333c9687df63377efceddd82efa9101913e8e
```

输出结果 8-17 查看黑名单 serial 和 pubkey

## 密钥和私钥管理

对于任何一种安全基础设施来说，如何存储秘密信息（对称加密中使用的密钥和“公-私钥”对中的私钥）都是一个需要严肃对待的课题。如果我们假设文件的访问权限限制对于保护这些秘密来说，已经足够强大了，那么就只需把这些秘密放在一个文件中，并给这个文件设置相应的访问权限就可以了。只可惜，Linux 底层的文件访问权限的灵活性较差，而且一旦设置有误，就会有秘密信息泄露的风险。此外，如果攻击者获取到了 root 权限，就能避开所有的权限限制，拿到所有他想要得到的东西。

Android 提供了一个名为 keystore 的服务，用户可以通过它访问秘密信息。在第 4 章<sup>1</sup>中我们已经讨论过这个服务了。keystore 这个服务以每个用户 id 为单位，将各个用户/应用的秘密信息分别存放在各个用户各自对应的/data/misc/keystore/user\_##目录中。但用户是不能直接访问这些目录的，而是只能通过 keystore 来访问它们——因为这些目录的权限设置是 0700，而它们的唯一拥有者是 keystore。keystore 服务提供了一些公开函数，如 generate、sign 和 verify 来提供密码学相关服务，但就是不允许应用/用户直接访问其中存储的私钥等秘密信息。这一设计使得 keystore 这个服务可以用硬件来实现。在 Android M 版中，将这一服务进一步扩展为 gatekeeper 服务，这个服务（详见第 5 章）把 password 替换为了一个 opaque handle，这一改动除非是用在硬件实现的加密机制上的，否则是没什么意义的。

事实上，从 JellyBean 版开始，Android 就已经提供了完全由硬件实现的秘密信息存储器（secure storage），并已经出现在了支持它的一些移动设备上了。在第 2 本的讨论中，我们可以看到：keymaster 硬件抽象层即提供了统一的加/解密操作接口，允许它既可以用硬件实现，也可以由软件来实现。这样就既能让安装了相应硬件的移动设备，实现一个完全由硬件实现的 keymaster 模块，也能让那些没有安装相应硬件的移动设备使用一个 softkeymaster。

完全由硬件实现的秘密信息存储器扩展了 ARM 处理器中 TrustZone 的用途。TrustZone 现在已经是所有 ARM 处理器的标准配置了，我们可以把一个“可信执行环境”（TEE，Trusted Execution Environment）加载在 TrustZone 中——这样，这个 TEE 就处于一段独立的、Android 不能直接访问其中内容的内存镜像中了。在被加载之后，Android 只能通过一条只能在内核（即，Supervisor）模式中调用的特殊指令（SMC，也就是 Secure Monitor Call）“切换进” TEE。在系统启动的过程中，一段指定的代码会被加载到 TEE 中（至于这段指定的代码具体是什么，则可以由厂商来指定），尽管谷歌在它自己“值得信赖的”操作系统中，已经提供了一段标准的基本实现，并在其中提供了 keystore 和 gatekeeper 的接口。我们将会在第 2 本中讨论它们。

---

1 原文如此，显然是第 5 章之误。——译者注



## 8.5 存储安全

### 加密/data 分区

尽管大多数用户并不在乎他们的手机中是不是使用了加密机制，但所有用户都会在不同程度上担心因为手机不小心被弄丢了或者被盗了，而导致个人隐私数据泄露。iOS 系统从 iOS 4 开始就提供了透明的加密服务，无独有偶，Android 系统也早从 Honeycomb 版开始，就引入类似的加密机制了。Honeycomb 版中通过使用 dm-crypt 机制（相同的机制也在 OBB 和 ASEC 中使用），把加密这个概念扩展到了整个文件系统层上。不过“全盘加密”（full disk encryption）这个术语用在这儿，还是多少有些不严谨的，因为被加密的通常只有/data 分区。相对于加密整个硬盘/闪存，这一做法其实更有意义——因为/system 分区中不会存储任何敏感数据（如果加密它的话，还会因为解密操作，使系统的性能受到很大的拖累）。

Android 开发者文档（网站）中给出了一份关于这一加密机制的详尽的文档（该文档经修订适用于 Android L 版）。和 OBB 与 ASEC 一样，卷管理器（Volume Manager）既要负责文件系统的加密，也要负责文件系统的解密。只不过前者是在用户选择需要执行相关操作时执行的，而且是一个相当冗长的操作，而后者则是在系统使用 device mapper 把加密文件系统作为一个块设备（block device）mount 上来时，自动执行的。

注意：文件系统加密和 OBB 与 ASEC 有个显著的区别：OBB 与 ASEC 的解密密钥是以明文的形式藏匿在系统的某个角落里的——只不过是只有 root 才有权读取它们罢了。而/data 分区的加密密钥却不能放在手机里，而是需要在系统启动时，通过与用户进行交互，请用户输入密码（或者更准确地说，是根据用户的屏幕解锁密码推导出/data 分区的加密密钥来的）。这就需要对 Android 系统的启动流程以及 init 和 vold 之间的交互操作进行修改——这些我们在第 4 章中都已经讨论过了。

在采用 dm-crypt 方案之前，其实还有人提出过其他几个文件系统加密备选方案（其中最著名的是由 Wang 等人提出的 EncFS<sup>[9]</sup>）。但 dm-crypt 已经成了事实上的标准，现在在 Android L 版中也已经默认使用它了。dm-crypt 的架构如图 8-5 所示。

Android M(PR1)中进一步给 dm-crypt 增加了一个名为“adoptable storage”的新特性。它使用户能进一步把 Android 的文件系统加密功能用在扩展存储（即 USB 存储设备）上。一般来说，这一功能是由 vold 来进行处理的，vold 会把加密密钥记录在/mnt/vold 中，并把解密出来的卷 mount 在/mnt/expand 目录里。

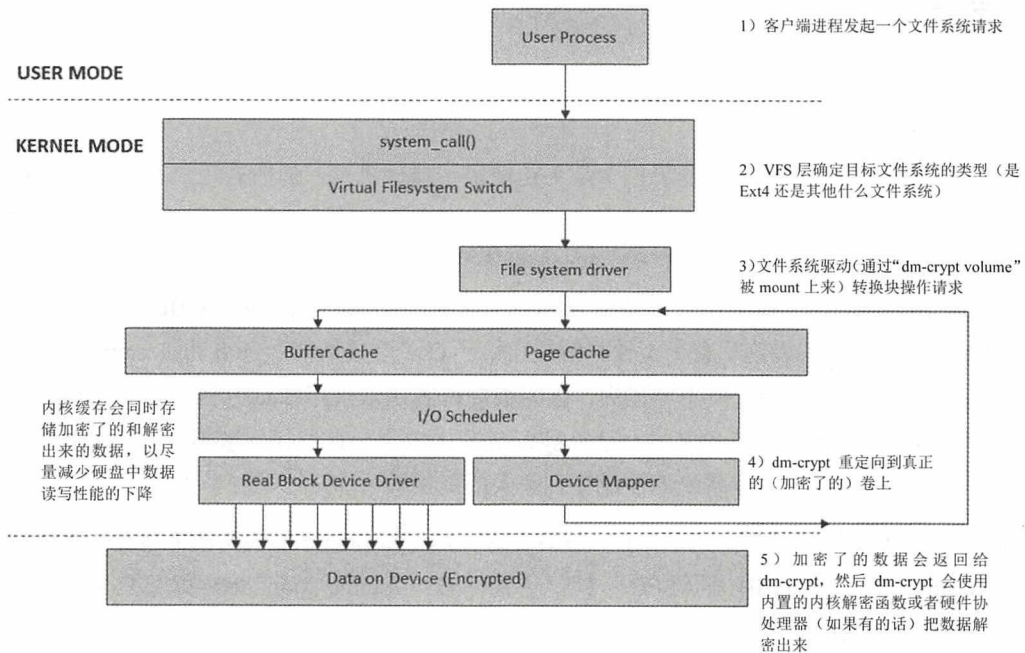


图 8-5 dm-crypt 的架构

## 与安全相关的注意事项

和其他许多解决方案一样，Android 中使用的这一解决方案也是一个 Linux 的通用解决方案。这也就是说：它并不是专为 Android 设计的。因此尽管它当前已经达到了它的所有设计目标，但它还是有可能不像一个专用的解决方案那样安全。

dm-crypt 解决方案（的安全性）依赖于一个主密钥（master key），在设备第一次启动时会随机生成这个 128 位的主密钥，然后这个主密钥又会被加密起来，加密它用的是一个默认口令（这个默认口令就是字符串“default password”<sup>1</sup>）再外加一个会被存储下来的 salt。然后，这个主密钥会被用来加密整个磁盘。当用户更改他的 PIN 码或口令（或者图形锁）时，就会用用户的 PIN 码/口令重新对主密钥做一次加密（主密钥本身不变），只有到了这个时候，没有用户的 PIN 码/口令，才没有办法获得这个主密钥。

就像对付大多数加密方案的套路一样，一个拥有足够资源的攻击者完全可以对它进行暴力攻击，或者挨个尝试字典中所有可能的 PIN 码或口令的组合。由于 dm-crypt 使用的主密钥实质

<sup>1</sup> “default password”就是英语中“默认口令”的意思。——译者注

上就是用用户的 PIN 码/口令/图形锁加密的, 所以攻击者只要反复调用 `vold` 自身代码中的 `cryptfs_check_passwd()` 函数 (这个函数会在 `mount` 文件系统之前, 用用户提供的口令进行解密操作, 检查 `dm-crypt` 尾部的数据是否正确, 这样就能判断出用户提供的口令是不是正确口令了), 就能逐个验证所有可能的口令组合——就这么简单! 如果用户使用的是 PIN 码或者图形锁, 而不是一个较长的混合使用了数字、大小写、特殊字符的口令, 那么拥有足够资源的攻击者就能很轻松地在几分钟甚至几秒内试遍所有可能的组合, 提取出主密钥来。

为了增加这种攻击的难度, Android 还将设备可信执行环境 (TEE, Trusted Execution Environment) 整合了进来, 一旦启动这一功能, 就会有一个存储在 TEE 中的次级密钥, 参与到对主密钥的加密和解密的过程中。在这种情况下, 理论上就能限定: 暴力攻击只有在本机上才能进行。

事实上, 这一解决方案相对于 Android 的竞争对手所使用的方法仍是极为相形见绌的。在 iOS 中同时使用了基于硬件的密钥存储和对预定义的不同文件类型使用不同密钥的方案——这是一种完全针对移动设备专门设计的完全彻底的解决方案。这一解决方案的有效性在 2016 年 2 月得到了有力的证明: FBI 试图迫使苹果公司提供解密涉案 iPhone 手机中数据的暴力攻击方法——但是从来就没有人向谷歌公司提出过类似的要求, 这就意味着对涉案的 Android 设备进行暴力攻击其实并不是一件特别困难的事。事实上, 对于一名拥有足够资源的攻击者来说, 如果能拿到 Android 设备, 特别是在 Boot Loader 没有加锁或者设备上运行的系统中有其他安全缺陷时, 对它进行暴力攻击仍然是具有可行性的。不管是在什么操作系统上, 也不论它是运行在移动设备上、台式机上还是服务器上的, 唯一真正管用的安全之道就是: 使用尽可能长的、不和其他口令重复且混合使用了数字、大小写、特殊字符的口令。

## 对性能的影响

一个经常被提及的问题是: 加密会对系统的性能产生多大的影响。从本质上说, 加密必然需要更多的 CPU 时间去完成解密和重新加密数据的操作, 这一定会影响系统的性能, 而且也更加耗电。不过在各种因素的作用下, 总的来说, 加密对系统性能的影响应该说是在微不足道到可控之间的, 相关的作用因素罗列如下。

- **访问存储设备本来就很慢了:** 尽管不像传统机械硬盘那么慢, 但是和 CPU 比起来, 闪存设备明显还是龟速的。尽管加/解密函数还会把每次访问所需的时间再拖慢上几微秒, 但是如果我们冷静地看一下各种拖慢存储设备读写速度的因素, 及其所导致速度变慢的百分比, 我们就会发现, 加密导致的速度变慢, 充其量也只不过和数据存储碎片化相当而已。
- **Linux 内核已经使用缓存对访问做了优化:** 如图 8-5 所示, Linux 内核已经用过缓存存储



设备上的数据的方式，优化了数据访问方式。因为 dm-crypt 显示为一个块设备，它位于缓存层的下方，所以就可以利用这一点做点文章：数据只需要解密一次，然后所有对该数据进行的读/写操作，就直接对缓存中（已经解密出来）的数据进行就可以了。只有当数据需要刷回到下方的设备中去时，它才会被再次加密回去，然后再写回到更底层的物理闪存设备中去。

- 从根子上说，对/data 分区的访问并不像对/system 分区的访问那样频繁：Linux 系统对存储着 Android 的各种框架和静态配置信息的/system 分区的访问是非常频繁的。相对而言，对/data 分区的访问就少多了——只有在要加载某个 App 或者偶尔在某些“运行时配置信息”需要变更时，才会要访问/data 分区。

## 基于文件的加密（Nougat 7.1）

Android 透明的基于文件系统的加密有好的方面，但也有坏的方面。一旦设备被解锁之后，文件系统就会被 mount 上来，从运行在该设备上的代码的角度看来，加密就像是不存在一样。这也就意味着，即使用户再次锁掉了设备，恶意 App 也仍然能访问用户的敏感数据。

在 7.1 版中，Android 更进一步使用了基于文件的加密技术，在这种加密模式下，将会给每个文件都分配一个必须用用户的 passcode 推导出来的密钥，通过使用 Linux 内核中的 EXT4\_FS\_ENCRYPTION 特性，再辅以 Keymaster 和硬件（即，TrustZone）的支持，特定的文件在屏幕被锁定之后，直到用户下一次解锁屏幕这期间就不能访问了。现在，Android 终于追平了 iOS 中提供“NSProtectionClass”机制的“cprotect”特性了。

## Direct Boot（Nougat 的新特性）

Nougat 版中引入了一个名为 Direct Boot 的重大改进。这一特性在利用 dm-crypt 的问题上与前文中描述的方法有一些反其道而行之的味道：它把用户加密的数据（Android 将其称为“credential encrypted”<sup>1</sup>）和设备中用硬件存储密钥和加密的其他数据（这些数据也被 Android 称为“device encrypted”）区分开来。后者是不安全的——因为设备加电之后，它就可以被访问到。但这也让 App 能在设备启动（或重启）后，马上就能开始运行，而不必等到用户解锁设备屏幕之后才能运行。

---

1 登录凭据加密，因为这些数据的解密密钥，是用你的登录凭据（口令/PIN 码/图形锁……）加密后存放在系统中的，要想拿到解密密钥，必须要用户登录（即，用户输入了登录凭据）之后才行，故得此名。——译者注

Direct Boot 设计的目的就是让一些涉及用户敏感信息，又需要随机启动的 App，能够在设备启动之后，用户解锁设备屏幕之前，就能接收消息，发出通知，并访问一些不敏感的数据。要使用 Direct Boot 的 App 必须在它的 manifest 文件中声明一个新的 broadcast receiver，用以处理 LOCKED\_BOOT\_COMPLETED 广播消息，并将 android:directBootAware 设为 true。

## 启动过程中加强验证

KitKat 中引入了一个新特性，利用内核的 device mapper，增强了启动过程的安全性。这个被称为 dm-verity 的特性是源自 Chromium 操作系统的，从 3.4 版本起，被并入了 Linux 内核（因而也就被并入了 Android 中）。从 Nougat 版开始，这个特性被设为默认强制启用。

回忆一下第 3 章的内容，在那里我们讲到过一个信任链[也就是“可验证启动路径”(verified boot path)]，这条信任链是从 ROM<sup>1</sup>开始，经过 Boot Loader，再到内核和 root 文件系统（也就是 boot 分区）这样一路构建起来的。尽管 Boot Loader 确实也应该去验证/system 是不是遭到过修改但它只有在重刷整个/system 分区时才会去做这个验证，这就给那些拥有 root 权限的进程（比如在系统被 root 了或者中了病毒时）对/system 做永久性的修改打开了方便之门——它们只要用读-写权限重新 mount 一下/system 分区，就能修改其中的内容了。使用 dm-verity 有效地将启动时的信任链再向前延伸了一段——将/system 也纳入其中。

要验证一个分区中的数据有没有被修改过，只要计算分区中的所有数据块的 Hash（DM-Verity 用的是 SHA-256 算法），然后把结果和事先存储的经过数字签名的 Hash 值相比较就可以了。可是这个做法的主要问题在于：读取整个分区中的所有数据块是要花很长时间的——这会让启动过程变得异常冗长。为了解决这个问题，dm-verity 只读取分区一次，并且把每个数据块（大小为 4KB）的 Hash 值记录在一颗树的叶子节点上，然后多个叶子节点又会被重新计算出一个 Hash 来，记录在树的第二级节点上，然后多个第二级节点又会再算一个 Hash 出来，记录在树的第三级节点上，……，直到在根节点上得到整个分区的 Hash 为止——这就是所谓的“根 Hash”（root hash）。随后，这个 Hash 会用厂商的私钥加以数字签名并保存，在这之后，就可以用厂商的公钥来验证这个根 Hash 有没有被改动过了。由于磁盘上数据的读取操作是以数据块为基本单位的，所以我们完全可以加上这样一个验证环节：读取到相关数据之后，先对已经在内核缓存/页缓存中的数据做一次 Hash 校验，只有在 Hash 验证通过的情况下，才会把读取到的数据返回给请求这些数据的进程；如果 Hash 验证不通过，我们就知道数据块中的内容已经被修改过了，这时就会报一个 I/O error 的错。而 Hash 本身是存储在 root 文件系统里的，它是（和

---

1 原文如此，但是你回顾一下第 3 章“Boot Loader 的加锁与解锁”一节，就会发现，这条信任链是从 rpm 到 sbl 再到 Boot Loader……的，可能作者把“RPM”误写成“ROM”了。——译者注

内核一起) 经过数字签名后, 被放在设备的 boot 分区里的。

dm-verity 的这一特性被誉为: 能够防止设备被安装恶意软件——因为它能有效地阻止对 /system 分区做任何修改, 同时还有个不错的副作用, 即能阻止各种未经授权的永久性 root。恶意软件确实能够对 /system 分区进行修改, 但 Android 也能检测出这些修改, 并可以在检测到修改后拒绝启动——这同样也能阻止各种“永久性 root”的后门(比如设置 /system/sbin/su 拥有 SetUID 的权限)。从厂商的角度看, 这个特性也很不错, 因为大多数厂商只提供 Boot Loader 解锁后的 root 方案, 而 Boot Loader 解锁本身就破坏了这个信任链的第一层。再进一步说, dm-verity 只需要对系统升级过程(这一过程详见第 3 章)做一些微调就可以了——即, 只有在系统升级的过程中, 厂商需要对被修改完毕后的 /system 重新做一个(根 Hash 的)数字签名; 而在设备整个生命周期的其余时间里, /system 分区一直是只读的, 所以它的验证签名也无须做任何修改。

dm-verity 在内核模式中的实现代码相当小——实现这个驱动程序的 /dm/dm-verity.c 文件的大小只有 20KB, 它被插入在 Linux Device Mapper 中(将在第 2 本中讨论它)。谷歌在 Android 文档<sup>[10]</sup>中详细说明了系统启动的过程。Android Explorations 博客<sup>[11]</sup>中还提供了进一步的细节信息, 甚至包括了如何在镜像的 build 过程中使用 veritysetup。

Android 的 root 程序很快就对 dm-verity 做出了回应, 现在已经提供了一种“无需修改 /system 分区的 root 方案”(systemless root), 在这一方案中, 设置 su 之类文件的权限时, 已经不再需要对 /system 分区进行修改了。这严重影响了 dm-verity 的有效性, 因为它允许设备被永久性地 root 掉, 而且同时还不会向用户发出任何提示或通知。接下来我们就来讨论 root 这个话题。

## 8.6 Root Android 设备

大多数厂商会在设备中保留 adb 这一功能, 让开发者能对设备进行必要的操作, 但是几乎没有厂商会让用户能以 root 权限去操作设备(我很怀疑真的会有这种厂商吗?)。不留 root 访问接口的理由非常充分: 我们知道, 如果能获取一个 UNIX 系统的 root 访问权限, 事实上就能在里面为所欲为——对 Android 来说, 也是一样的。在系统中留有 root 权限的访问接口还可能会给恶意软件留下可乘之机——而我们知道在 Android 的世界里还真就不缺恶意软件。拥有 root 访问权限, 也就意味着可以读系统中的任何文件, 甚至更糟——可以修改或删除系统中的任何文件。这会让攻击者既能拿到设备用户的个人隐私数据, 又能获得对设备的控制权。

对于苹果的 iOS 来说, 上述事实也是成立的(因为 iOS 也是一种 UNIX 系统, 只不过是基于 Darwin 的而已)。不过 Android 和 iOS 在这个问题上的差别还是比较大的: 苹果的开发者的从头到脚重新创建了这套系统的(对, 还确实就是从头到脚), 从最基本的硬件到最顶层的软件,



所以他们能够把所有的东西整合成铁板一块，并且不惜一切代价地阻止 root 权限的访问（事实上，是不允许任何超出应用沙箱模型权限范围之外的访问）。Android 则是架构在 Linux 之上的，而 Linux 本身就是由许多贡献者开发的代码组成的大杂烩——并不是所有的代码都是按最严格的安全标准开发的。另外，有些厂商也会留下后门（让系统能够启动到另一个可选的配置下），而任何可能拿到手机一段时间的人都能利用这一点获取 root 访问权限。从另一个角度，或许能把事情看得更清楚：Android 是把应用当贼防的，而 iOS 却是把用户当贼防的。

## 在设备启动环节中 root

当 Android 设备启动时，启动流程通常都做了哪些操作，我们在第 3 章中都已经讨论过了。不过，在进入安全模式、系统升级或者进入 recovery 模式时，系统启动过程会被转移到另一个可选的配置上去。这一般需要你在设备启动时，按住某个指定的实体键的组合键 [通常是按住某个音量键或者同时按住音量增加/降低键，再按下 home 键（如果设备上有 home 键的话)], 或者用 USB 线把移动设备和电脑相连，并在电脑上运行 fastboot 命令。一旦启动流程被转移之后，就可以引导 Boot Loader 去加载另一个可选的 boot 镜像——这个镜像既可以是闪存上的 recovery 镜像，也可以是 SD 卡中的升级包，或者是（通过 USB 线）由 fastboot 传来的镜像。

如果一台设备的 Boot Loader 可以被解锁（详见第 3 章），那么这台设备就可以被 root。这很简单，前文已述，解锁 Boot Loader 时，/data 分区将会被格式化——以防用户的敏感数据因此而落入不该拿到它们的人的手里。另外，有些 Boot Loader 还会设置一个永久性的标志位，表示 Boot Loader 已经被修改过了——哪怕后来 Boot Loader 又被重新加锁了之后，这个标志位也不会被改回去或撤销。必须要注意的是：一旦 Boot Loader 不再强制检查要被刷入移动设备的镜像的数字签名，那么它基本上也就没法对系统的安全负责了。

这时“root”移动设备，真的很方便！我们只需要去修改设备镜像的一个组成部分 init RAM disk (initramfs) 就可以了。由于内存会把 initramfs 作为根 (root) 文件系统 mount 上来，并以 root 权限启动其中的 /init 二进制可执行文件，所以我们只要换上一个改过的 /init 文件，甚至只需换上一个修改过的 /init.rc，就足以获取 root 权限了。从这一点出发，可以更进一步直接让 ADB 保留 root 权限（通过设置系统属性 ro.secure=0<sup>1</sup>）或者直接换上一个不会放弃 root 权限的 ADB，这样就能让我们以后能很方便地以 root 身份访问该设备了。不过大多数 root 工具通常会把一个名为“su”的二进制可执行文件放到 /system/bin 或者 /system/xbin 目录中去，然后再用命令“chmod 4755”设置这个可执行文件的 setuid 位。这样当用户在 shell 中调用这个程序是，setuid 的作用

---

1 在最新的一些 build 版本中，adb 是条件编译的 (#ifdef ALLOW\_ADB\_ROOT)，设置了 ALLOW\_ADB\_ROOT 这个条件编译开关后，就完全可以忽略掉这个系统属性。——原注

就会发挥出来——让系统自动授予它 root 权限。（在 KitKat 版之前）编写这样一个 su 程序非常简单，甚至可以把它浓缩成三行代码，如代码清单 8-5 所示。

代码清单 8-5 在未强制启用 SELinux 的设备中，简化了的 su 程序的实现代码

```
#include <stdio.h>
void main(int argc, char **argv)
{
    setuid(0);
    setgid(0);
    system("/system/bin/sh");
}
```

你可以在 AOSP 的 `/system/extras/su/su.c` 中找到一个类似的例子（只不过这个例子是支持命令行参数的）。不过在 KitKat 及以后的版本中，由于引入了强制启用的 SELinux，使得这个二进制可执行文件变得更加复杂了——因为这时它的父进程（也就是 shell）已经被限制在一个受限的执行上下文环境（`u:r:shell:s0`）中，这是 su 所不能改变的事实。为了仍能获取 root 权限，su 程序就需要与某个处于 `u:r:init:s0`（或 `u:r:kernel:s0`）状态的、不受限制的上下文环境中的进程，进行一次 IPC（进程间通信）调用，然后再 fork 出一个 shell 来 [比如，WeakSauce 漏洞利用代码（及 DaemonSu）可获取 root 权限，在本书的官网上对此有详细的解释<sup>[12]</sup>]

如果你有一台安装了 KitKat 或更高版本的系统的移动设备（KitKat 及以后版本的 Android 系统中均强制启用了 SELinux），且该设备已经被 root 了，那么你可以亲自动手，看到类似输出结果 8-18 所示的信息。

```
#
# Starting off with a non-privileged shell, get PID and UID:
shell@htc_m8wl:/ $ echo $$
6498
shell@htc_m8wl:/ $ id
uid=2000(shell) gid=2000(shell) groups=1003(graphics),1004(input),1007(log),1009(mount),
1011(adb),1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admin),3002(net_bt),3003(inet),
3006(net_bw_stats) context=u:r:shell:s0
#
# Switch to a root shell, do same (i.e. get PID and UID):
shell@htc_m8wl:/ $ su
root@htc_m8wl:/ # echo $$
6503
root@htc_m8wl:/ # id
uid=0(root) gid=0(root) context=u:r:init:s0
# Use the toolbox specific -Z flag to ps, to show SELinux contexts
root@htc_m8wl:/ # ps -Z
# Note the "su" is the child of the shell (in this case, 6498), but has no children itself.
# the actual shell spawned by su is started from a daemonsu instance, which gets the u:r:init
# unrestricted SE-Linux context from daemonsu. eu.chainfire.supersu is the GUI app.

u:r:shell:s0      shell      6498  601  /system/bin/sh
u:r:shell:s0      shell      6503  6498  su
u:r:init:s0       root       6506  5319  daemonsu:0:6503
u:r:init:s0       root       6510  6506  tmp-mksh
u:r:untrusted_app:s0  u0_a140   6528  575   eu.chainfire.supersu
u:r:init:s0       root       6578  6510  ps
```

输出结果 8-18 观察一个能绕过 SELinux 获取 root 权限的 su 的实现方式



在现实生活中，由于 root 移动设备的做法是如此普遍，竟然催生出了不少“SuperUser”应用。这些应用能在设备被 root 之后，提供一个方便管理哪些 App 可以拥有 root 权限的 GUI 接口。这些应用实际上是提供了一个 API（通过 permission 和 intent），使其他应用能够获取 root 权限。Chainfire 的 SuperSU 就是一个挺有名的例子。这个应用定义了它自己的 Dalvik 层级上的权限（android.permission.ACCESS\_SUPERUSER 和 eu.chainfire.supersu.permission.NATIVE），使应用能够通过广播 intent 的方式得到超级用户的特权。这个应用也能和你在上面这个输出结果中看到的那样，通过一系列巧妙的操作绕过 SELinux 的限制。

## 利用安全漏洞 root

无论厂商是不是留下了设备启动时 root 的后门，系统中常常还会有其他漏洞存在。而且不像前者，这些漏洞都是无意间被留下的，它们可能造成的危害全看这些系统漏洞的利用方式。安全漏洞的利用方式五花八门，千奇百怪，而且在被发现之前通常是很难预测的，但是它们仍有一个共同的特征：找到一些不安全的配置或软件组件，进而让攻击者指定的一段代码（shellcode）被执行，通过这一方式，最终获取 root 访问权限。在本章开头的“移动安全威胁建模”一节中，我们已经提到过，对这类攻击有个专门的安全术语：提权攻击（privilege escalation），因为这类攻击一般是从一个低权限的进程（比如某个应用）着手，然后不断提升它的权限，一般是先提到 system user 的权限，并最终提升到 root 权限。

基于漏洞的 root 方法和 iOS 中的“越狱”十分类似。它们都需要发现和利用软件中的 bug。在一个完美的世界里（至少是在完美的谷歌和苹果的世界里）是不应该有这两种（root/“越狱”）方法存在的。一旦这些 root/“越狱”被披露之后，它们能有效工作的日子也就屈指可数了：操作系统会马上推出安全补丁，并且建议用户下载并更新（有些厂商，比如亚马逊的 Kindle，甚至会自动更新）。其中一个较为有名的例子是 Android 的 Gingerbread 版，在发现（这一版 Android 系统使用的 Linux 内核中存在一个漏洞）可能会被恶意软件频繁地利用时，谷歌（甚至等不及 Linux 推出相应的补丁）就自己推送了一个 Linux 内核安全漏洞的修复补丁。

对漏洞利用技巧做一个详尽的讨论显然已经大大超出了本书的范围，而且坦率地说也没什么意义——因为当前所有已知的漏洞已经全部被修补掉了。在漏洞利用过程中，获取 root 权限的方法通常都是：向某个以 root 权限运行的进程（vold 是黑客的真爱……）输入一段精心构造的数据，破坏其内存结构（栈溢出或堆溢出），然后通常是覆盖某个函数的指针（或者，更常见的，覆盖某个返回地址），以破坏目标进程的执行流，并把它重定向到攻击者能够控制的输入数据上。另一种名为 ROP（Return Oriented Programming，返回导向编程）的技巧也经常会被使用，在进行这种攻击时，攻击者会有意识地把执行流重定向到目标进程中已有的代码片段上去——不过是以攻击者控制着的方式执行的。这种方法有点类似于生物学中的 DNA 拼接和重组，用



它可以绕过 ARM 的 XN bits 之类的数据执行保护措施。对已知的各种漏洞利用方法及 ROP 技巧的更详细讨论，请参考 *Android Hacker's Handbook*<sup>1</sup> 一书。

值得注意的一点是：并非所有的漏洞利用技巧都必须涉及代码注入——有些漏洞的利用方法甚至更简单也更漂亮（比如针对 HTC One 手机的“WeakSauce”漏洞的利用技巧，详见本书官网<sup>[12]</sup>）。同样，还曾经出现过一个经典的漏洞（它并不是针对 Android 的，而是针对 Linux 内核的），可尽管是这样，Android 仍然躺枪。它就是 Geohot 发现的十分巧妙的“Towelroot”漏洞利用方法<sup>[14]</sup>。这个漏洞利用 Linux 内核在处理 fast mutexes 时的一个有名的 bug（CVE-2014-3153），能够获取 root 权限。尽管“Towelroot”本身并不是一个恶意软件，而只是一个能用来 root 系统的实用程序，但是恶意软件也能利用完全相同的方法，无须用户同意，在用户不知不觉中，悄悄地把 root 权限弄到手。

借用 Donald Rumsfeld 的一句名言——上述这些漏洞是 Android 世界中的“已知的未知”，它们实质上是一些过去未知的，但现在已经被发现并被修补掉了的 0day 漏洞。但在 Android 世界中还有“未知的未知”。后者是指那些可能会存在的，但还没有被发现的漏洞，或许可能更糟，这些漏洞已经被发现了，但至今还未被披露。任何被黑客发现的 0day 漏洞，实质上都是进入所有存在这个特定漏洞的 Android 设备的一把万能钥匙。恶意的黑客可以把漏洞整合进一个功能强大的恶意软件中；或者如果不想这么麻烦的话，也可以直接在市场上把它卖掉。尽管不像 iOS 的漏洞利用代码这么值钱，但是一个 Android 0day 漏洞在任何地方都能卖上 5 万到 50 万美元不等的价钱 [实际价格还要参考漏洞的攻击条件（是远程还是本地）以及对系统的影响而定]。

“Dirty COW”漏洞（CVE-2016-5195）可能是对 Android 所面临的严峻的安全形势的最好注解了。这个 bug 是发生在 Linux 内核处理“写时复制”内存页的核心代码中的（它的存在时间甚至比 Android 的年龄还大）——Linux 世界中真正的“未知的未知”，在 2016 年 10 月被披露之前，它已经静静地存在了十几年。这个漏洞能让任何进程立即把它的权限提升到 root 级别——可以用来 root 你的设备，这对于 Android 的安全体系来说，完全是一场灾难。如果你是在 2016 年或者 2017 年阅读到本书，和本书第一次出版时相比，出现的新变化就是：你的移动设备在这个漏洞面前是非常脆弱的——一个漏洞利用方法已经传开了，但漏洞的补丁却还没有影儿……

## Root 对安全的影响

由于基于 boot 的 root 方案需要用户干预，并连接电脑，所以一般并不认为它是 Android 系

---

1 该书的中文版为《Android 安全攻防权威指南》，诸葛建伟、杨坤、肖梓航译，人民邮电出版社，2015 年 4 月出版。——译者注

统中的一个不安全因素。但无论如何，它都给了能够拿到手机的攻击者一个可乘之机。如果设备丢了，被盗了，或者只是离开了用户视线之外一段足够长的时间，就会造成问题。一个熟练的攻击者只需要花十几分钟就能 root 一台手机，从中窃取出所有的用户个人信息，并留下一两个后门。这也就是为什么大多数的 Boot Loader 通常都是加锁的，而且一旦发现 Boot Loader 可能被解锁，就会强制恢复出厂设置并擦除所有用户个人信息的原因——因为一旦 Boot Loader 被解锁，设备就存在安全风险（除非 Boot Loader 再次被加锁回去）。

同样是 root，利用漏洞攻击的方法则简单得多。root 时，不需要用户手工改变系统的启动流程，事实上攻击过程中甚至都不需要用户的干预。这给 root 自己手机的用户带来了便利（想想那些“一键 root”的工具吧），但它也隐藏着巨大的风险：root 可能在用户不知不觉的时候就已经完成了。比如，通常的攻击方式是：诱使用户安装一个看似无害的 App，App 中携带的特洛伊木马就把整个系统给拿下了。

如果能用来 root 设备的漏洞是与 HTTP 有关的话，那么危险还会进一步增加：当漏洞的利用（部分）涉及浏览器时，只要诱使受害者访问一个恶意的网页或者让受害人不经意间访问网页中的一部分内容（比如在网页上放一个广告），恶意代码就会在目标浏览器上运行，并在设备上获得一个立足点，在接下来的攻击步骤中，含有更多恶意功能的更厉害的恶意代码就会被注入进来，先是获取远程执行代码的权限，最终能够远程 root 目标设备。

下面还要说一下的是，如果是去不可信的网站下载的 root 工具，那么 root 设备这个操作本身就是很危险的：当某个用户急不可待地去下载 root 工具（无论是一键 root 工具，还是需要用户人工干预才能完成 root 的工具），想要尝尝鲜时，如果下载源不是绝对可信的，是很难（事实上是不可能）检测出这个工具中有没有被注入额外的恶意代码或是后门程序的。缺乏合适的工具，也会在 root 的过程中，无意间对系统的二进制可执行文件或框架做了修改（比如无意间禁用了 Dalvik 的权限机制），而这也给恶意软件以可乘之机。恶意软件也可能在 Linux 内核底层注入一个 rootkit——尽管大多数情况下不会做得这么绝，而它只需在较高的层中，就能直接轻而易举地黑掉整个系统。有些讽刺的是，过去曾经发现有些 SuperUser 应用本身就有安全漏洞，能够让流氓应用检测到自己被装在了已经被 root 了的设备上，从而直接利用 SuperUser 这个应用把自己提升到 root 权限（参见 CVE-2013-6774）。

想要 root 自己手机的用户还有最后一个问题（而且这并不是最不重要的一个问题）需要仔细考虑清楚：对于应用来说，Android 对应用的数据（content）的保护机制在一台已经 root 了的设备上也是失效的——OBB 中存储的内容可以被 root 用户读取出来，因为 root 用户有权读取到 ASEC（Android 安全存储）的密钥。应用加密的数据也会遇到类似的情况，尽管被存放在硬件中的证书存储机制会给攻击者造成一些麻烦，但是应用进程自身的内存却很容易被 root 读取

到（通过 `ptrace(2)` 或其他类似的方法）。此外，DRM 机制也会不幸中刀。更惨的是：对于一个正在运行的应用来说，并没有一个简单有效的方法能检测出当前这个系统是不是已经被 root 了，进而拒绝在已经 root 了的设备上继续运行。也不是说这方面的努力就没有，谷歌 Play 的服务中就包括一个相当复杂的“safety net”机制，这个机制的设计目的就是在设备中执行多个运行时检测，看设备是否能够通过 Android CTS 兼容性测试——这些测试中就包括 root 检测。这一机制可以供应用，特别是 Android Pay 应用<sup>1</sup>使用，这样它们就能拒绝在已经被 root 了的设备（也很可能是已经被黑掉的设备）上运行了。可惜在实际操作中，尽管谷歌经常不断地大力度升级，但是每出一种新检测手段，都很快就会被 root 技巧绕过，比如文中提到过的“无须修改/system 分区的 root 方案”，它根本就不会在文件系统中留下可供检测的痕迹。

有人或许会争辩说，iOS 系统“越狱”之后不也一样吗？——确实，苹果的 fairplay 保护机制和应用加密（尽管比 Android 的要强些）在“越狱”面前是一样脆弱的。但是我们也应该记住 iOS 的越狱只有通过安全漏洞才能进行（而且随着 iOS 版本的升级，每次都比以前更难些），而大多数 Android 设备还允许在设备启动时进行 root，再加上 Dalvik 字节码十分便于反编译这一事实，这些都应该足以引起应用开发者的高度关注。

## 本章小结

---

本章试图引领你概览 Android 中大量的安全特性（无论它是继承自 Linux 的还是 Android 特有且大多数是实现在 Dalvik 虚拟机层上的）。我们特别提到了 Android 现在已经开始使用的 SELinux——尽管现在在使用 SELinux 时，尚有所保留，但是三星在 KNOX 中也采用了 SELinux，而且可以预计在最近将会发布的新版 Android 中，SELinux 将会扮演一个更为重要的角色。

尽管我们尽可能详细地讨论相关技术，但这并不意味着我们的讨论是十分全面的。有兴趣的读者应该进一步去阅读一些 Android 安全方面的专著，比如 Nikolay Elenkov 编写的 *Android Security Internals*<sup>2[15]</sup>——这本书中所有的章节都被用来讨论这一章各个小节中讨论的内容。

## 参考文献

---

- [1] a. SEAndroid - A Paper, Smalley/Craig:  
[http://www.internetsociety.org/sites/default/files/02\\_4.pdf](http://www.internetsociety.org/sites/default/files/02_4.pdf)
- b. SEAndroid - A Presentation, Smalley/Craig:

---

1 谷歌的电子支付应用。——译者注

2 该书的中文版是《Android 安全架构深究》，刘惠明、刘跃译，电子工业出版社，2016 年 3 月出版。——译者注



[http://www.internetsociety.org/sites/default/files/Presentation02\\_4.pdf](http://www.internetsociety.org/sites/default/files/Presentation02_4.pdf)

c. SEAndroid at ABS - Smalley/Craig:

[http://events.linuxfoundation.org/sites/events/files/slides/abs2014\\_seforandroid\\_smalley.pdf](http://events.linuxfoundation.org/sites/events/files/slides/abs2014_seforandroid_smalley.pdf)

- [2] Android Developer, SELinux: <http://source.android.com/devices/tech/security/se-linux.html>
- [3] RedHat, RHEL6 and SELinux: [https://access.redhat.com/site/documentation/en-US/Red\\_Hat\\_Enterprise\\_Linux/6/html/Security-Enhanced\\_Linux/](https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Security-Enhanced_Linux/)
- [4] RiskIQ, Mobile Apps on Google Play: <http://www.riskiq.com/company/press-releases/riskiq-reports-malicious-mobile-apps-google-play-have-spiked-nearly-400>
- [5] a. BlueBox, Android "Master Key Vulnerability": <https://bluebox.com/technical/uncovering-android-master-key-that-makes-99-of-devices-vulnerable/>  
b. Saurik, Android "Master Key Vulnerability": [www.saurik.com/id/17](http://www.saurik.com/id/17)
- [6] BlueBox, Android "Fake ID Vulnerability": <http://bluebox.com/technical/android-fake-id-vulnerability>
- [7] Android Explorations, Certificate Pinning in 4.2:  
<http://nelenkov.blogspot.com/2012/12/certificate-pinning-in-android-4.2.html>
- [8] Android Documentation, Device Encryption:  
<https://source.android.com/devices/tech/encryption/index.html>
- [9] EncFS, by Wang et Al:  
[http://cs.gmu.edu/~astavrou/research/Android\\_Encrypted\\_File\\_System\\_MDM\\_12.pdf](http://cs.gmu.edu/~astavrou/research/Android_Encrypted_File_System_MDM_12.pdf)
- [10] Android Documentation, DM-Verity: <https://source.android.com/devices/tech/security/dm-verity.html>
- [11] Android Explorations, KitKat Verified Boot: <http://nelenkov.blogspot.com/2014/05/using-kitkat-verified-boot.html>
- [12] NewAndroidBook.com, Analyzing the WeakSauce Exploit:  
<http://newandroidbook.com/Articles/HTC.html>
- [13] Android Hacker's Handbook, Wiley 2014, by Joshua Drake and others
- [14] Towelroot.com, <http://www.towelroot.com>
- [15] Android Security Internals, No Starch 2014, by Nikolay Elenkov

过瘾了？如果本卷的内容引发了你进一步探索 Android 内部实现细节的兴趣，那么敬请期待

待本书的第 2 本吧！——马上来袭！<sup>1</sup>在第 2 本中，我们将会继续讨论本书中悬而未解的问题，讨论系统真正的内部工作机制（框架服务、图形、音频、多媒体，等等）——还是从程序员的视角出发的！

敬请不吝笔墨给我来信，让我知道你想读到什么，讨厌什么！也敬请经常留意本书的官方网站——[NewAndroidBook.com](http://NewAndroidBook.com)，那里会有更多的更新内容，海量的进阶资料！我们下一本见！



---

<sup>1</sup> 临近本书中文版交稿时（2016 年 12 月），本书作者 Johnson 向我（译者）亲口承诺：第 2 本将在 2017 年 3 月前出版！呃……可是谁知道呢？上次他承诺说，第 2 本会在 2016 年夏秋之交出版……——译者注



## 业界好评

这本书的确是目前一流的 Android 书。

——wushi (吴石), 腾讯科恩实验室负责人

一本对 Android 底层架构全面、深入剖析的书……帮助读者从细节上掌握每一个模块的要点。

——张鸿洋

……每个 Android 开发者都应该阅读一下这本书, 它会让你了解真正的 Android, 让你对 Android 底层系统有一个全新的认识。

——Stormzhang, 公众号: stormzhang

作者用“上帝”的视角, 让读者无须接触大量源代码就能了解整个 Android 系统的实现思想, 而这是比源代码更加重要的东西。相信读者在这本书的指引下一定会对 Android 系统有更加深入的理解和认识。

——徐宜生, 《Android 群英传》作者

本书可谓是了解 Android 系统内部技术的不二之选。

——段建华, 技术小黑屋 (droidyue.com) 博主, 公众号: droidyue\_com

注册成为博文视点社区 (www.broadview.com.cn) 用户, 即享受以下服务:

- 提勘误赚积分: 可在【提交勘误】处提交对内容的修改意见, 若被采纳将获赠博文视点社区积分 (可用来抵扣购买电子书的相应金额)。
- 交流学习: 在页面下方【读者评论】处留下您的疑问或观点, 与作者和其他读者共同交流。

页面入口: <http://www.broadview.com.cn/31813>



博文视点Broadview



@博文视点Broadview



策划编辑: 刘 皎  
责任编辑: 白 涛  
封面设计: 吴海燕

欢迎投稿

邮箱: Ljiao@phei.com.cn  
电话: 010-88254395  
新浪微博: @皎丫子

上架建议: 移动开发

ISBN 978-7-121-31813-9



9 787121 318139 >

定价: 89.00元